

«a2» 2013.11.11

Volume IV

os32

«

ISBN 978-88-905012-2-7

«**Appunti Linux**» -- Copyright © 1997-2000 Daniele Giacomini

«**Appunti di informatica libera**» -- Copyright © 2000-2010 Daniele Giacomini

«**a2**» -- Copyright © 2010-2013 Daniele Giacomini

Via Morganella Est, 21 -- I-31050 Ponzano Veneto (TV) -- appunti2@gmail.com

You can redistribute this work and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version, with the following exceptions and clarifications:

- This work contains quotations or samples of other works. Quotations and samples of other works are not subject to the scope of the license of this work.
- If you modify this work and/or reuse it partially, under the terms of the license: it is your responsibility to avoid misrepresentation of opinion, thought and/or feeling of other than you; the notices about changes and the references about the original work, must be kept and evidenced conforming to the new work characteristics; you may add or remove quotations and/or samples of other works; you are required to use a different name for the new work.

Permission is also granted to copy, distribute and/or modify this work under the terms of the GNU Free Documentation License (FDL), either version 1.3 of the License, or (at your option) any later version published by the Free Software Foundation (FSF); with no Invariant Sections, with no Front-Cover Text, and with no Back-Cover Texts.

Permission is also granted to copy, distribute and/or modify this work under the terms of the Creative Commons Attribution-ShareAlike License, version 2.5-Italia,

as published by Creative Commons at <http://creativecommons.org/licenses/by-sa/2.5/it/>.

This work is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

La diffusione dell'opera, da parte dell'autore originale, avviene gratuitamente, senza alcun fine di lucro. L'autore rinuncia espressamente a qualunque beneficio economico, sia dalla riproduzione stampata a pagamento, sia da qualunque altra forma di servizio, basato sull'opera, ma offerto a titolo oneroso, sia da pubblicità inserita eventualmente nell'opera stessa o come cornice alla sua fruizione.

L'opera, nei file sorgenti e nella composizione finale, include degli esempi in forma di sequenze animate, contenenti eventualmente anche delle spiegazioni vocali. Si tratta di video brevi e di qualità molto bassa. Tuttavia, a seconda di come viene diffusa o fruita l'opera, può darsi che sia necessario assolvere a degli obblighi di legge.

Il numero ISBN 978-88-905012-2-7 si riferisce all'opera originale in formato elettronico, pubblicata a titolo gratuito.

Se quest'opera viene consultata in-linea, attraverso uno spazio web, le informazioni generali sull'accesso ai file dell'opera (indirizzo IP di origine, nome del provider della connessione di origine, nome e versione del navigatore usato per accedere, geolocalizzazione dell'origine, data e orario di accesso), assieme al dettaglio delle pagine visitate o dei file scaricati, potrebbero essere annotate in un registro (log) di tale spazio web. Ciò per fini di controllo e statistica, a qualunque titolo. La conservazione di tale registro, se presente, dipende dalla politica del gestore e potrebbe avvenire per una durata di tempo indeterminata.

Quest'opera non contiene tecnologie atte a raccogliere e annotare i dati persona-

li degli utenti che la consultano. Tuttavia, lo spazio web che la ospita potrebbe richiedere una forma di iscrizione o di autenticazione. Se tale iscrizione o autenticazione fosse richiesta, è necessario valutare l'informativa sul trattamento dei dati personali dello spazio web in questione, per sapere come e a che scopo tali dati vengono trattati.

Quest'opera, così come realizzata dal suo autore, non contiene inserzioni pubblicitarie. Tuttavia, lo spazio web che la ospita potrebbe iniettare il codice necessario a somministrare della pubblicità durante la sua consultazione o prima dello scarico dei file. Tali inserzioni pubblicitarie, se ci sono, non hanno nessuna relazione con l'autore di quest'opera e nemmeno vi portano alcun beneficio economico, in quanto servono esclusivamente al mantenimento dello spazio web ospitante.

Le inserzioni pubblicitarie, se presenti, possono utilizzare una tecnologia atta a riconoscere gli accessi che provengono dallo stesso computer o dallo stesso terminale, assieme a tutte le informazioni che possono essere estrapolate dall'origine dell'accesso e sulle funzionalità del computer o del terminale usato per accedere (incluso il fatto che sia disponibile o meno del software che possa essere utile a recepire la pubblicità stessa). Per conoscere il modo in cui le informazioni vengono raccolte dalla pubblicità (se c'è) e il loro utilizzo effettivo, è necessario valutare l'informativa sul trattamento dei dati personali dello spazio web che ospita l'opera.

Una copia della licenza GNU General Public License, versione 3, si trova nell'appendice [A](#); una copia della licenza GNU Free Documentation License, versione 1.3, si trova nell'appendice [B](#); una copia della licenza Creative Commons Attribution-ShareAlike, versione italiana 2.5, si trova nell'appendice [C](#).

A copy of GNU General Public License, version 3, is available in appendix [A](#); a copy of GNU Free Documentation License, version 1.3, is available in appendix [B](#); a copy of Creative Commons Attribution-ShareAlike License, italian version 2.5, is available in appendix [C](#).

Per tutti i riferimenti dell'opera si veda <http://informaticalibera.net>. Al momento della pubblicazione di questa edizione, i punti di distribuzione in-linea più importanti, sono presso Internet Archive (<http://www.archive.org/details/AppuntiDiInformaticaLibera>), il GARR (<http://appuntilinux.mirror.garr.it/mirrors/appuntilinux/>), ILS (<http://appunti.linux.it>) e il Pluto (<http://a2.pluto.it>).

Parte vii	Progetto	9
83	Primi passi verso un sistema per hardware x86-32	11
84	Studio per un sistema a 32 bit	235
Parte viii	Manuale	441
85	Gestione	455
86	Sezione 1: programmi eseguibili o comandi interni di shell	467
87	Sezione 2: chiamate di sistema	497
88	Sezione 3: funzioni di libreria	603
89	Sezione 4: file speciali	755
90	Sezione 5: formato dei file e convenzioni	769
91	Sezione 7: varie	773
92	Sezione 8: comandi per l'amministrazione del sistema	777
93	Sezione 9: kernel	789
Parte ix	Codice	963
94	Script e sorgenti del kernel	967
95	Sorgenti della libreria generale	1789
96	Sorgenti delle applicazioni	2231
	Indice analitico del volume	2481

os32 è un piccolo sistema operativo di studio, realizzato come derivazione di os16, riutilizzando anche il lavoro su «05» (che si trova in appendice al volume). Molte cose di os16 sono rimaste praticamente identiche in os32, altre sono sviluppate in maniera differente

o semplicemente più completa. Tuttavia os32 rimane un sistema con una gestione fragile delle unità di memorizzazione di massa e del file system, per cui non ci si deve allarmare di fronte a un'improvvisa distruzione del file system in uno dei file immagine in cui il sistema si concretizza.

Le limitazioni di os32 comportano però una grande semplificazione nel codice, con l'intento di rendere os32 più utile, come primo approccio allo studio dei sistemi operativi, eventualmente anche prima di affrontare le pubblicazioni fondamentali, costituite da: *The design of the UNIX operating system*, di Maurice J. Bach, pubblicato da Prentice Hall, 1990, ISBN 0132017997 e *Operating Systems: Design and Implementation*, di Andrew S. Tanenbaum e Albert S. Woodhull, pubblicato da Prentice-Hall, 2006, ISBN 0-13-142938-8.

```
os32: a basic os
This is terminal /dev/console0
login:  root
password:  ****
```


83	Primi passi verso un sistema per hardware x86-32	11
83.1	Privilegi dei segmenti	15
83.2	Funzioni di utilità generale	22
83.3	Utilizzo dello schermo VGA a caratteri	25
83.4	GDT	27
83.5	IDT	46
83.6	Gestione delle interruzioni	64
83.7	Gestione del temporizzatore: PIT, ovvero «programmable interval timer»	73
83.8	Tastiera PS/2	77
83.9	Gestione di dischi PATA	79
83.10	PCI	112
83.11	NE2000	132
83.12	IPv4 in una rete Ethernet	182
83.13	Riferimenti	229
84	Studio per un sistema a 32 bit	235
84.1	Introduzione a os32	241
84.2	Caricamento ed esecuzione del kernel	256
84.3	Funzioni interne legate all'hardware, nei file «kernel/ibm_i386.h» e «kernel/ibm_i386/...»	274
84.4	Gestione dei processi	286

84.5	Caricamento ed esecuzione delle applicazioni	324
84.6	Gestione della memoria	332
84.7	Dispositivi	339
84.8	Gestione del file system	370
84.9	Gestione delle interfacce di rete	409
84.10	Gestione di IPv4	420
84.11	Gestione del protocollo ICMP	427
84.12	Gestione dei protocolli UDP e TCP	429

Primi passi verso un sistema per hardware x86-32



83.1	Privilegi dei segmenti	15
83.1.1	DPL, CPL e RPL	16
83.1.2	Pila dei dati	19
83.1.3	Segmenti, selettori e registri	19
83.2	Funzioni di utilità generale	22
83.2.1	Porte I/O	22
83.2.2	Sospensione e ripristino delle interruzioni hardware 24	
83.3	Utilizzo dello schermo VGA a caratteri	25
83.3.1	Memoria dello schermo a caratteri	25
83.3.2	Cursore	26
83.4	GDT	27
83.4.1	Privilegi	28
83.4.2	Organizzazione e contenuti della tabella GDT	28
83.4.3	Struttura effettiva della tabella GDT	30
83.4.4	Tabella GDT elementare	34
83.4.5	Costruzione di una tabella GDT	35
83.4.6	Attivazione della tabella GDT	38
83.4.7	Verifica della tabella GDT	40
83.4.8	Istruzioni per l'attivazione	42
83.5	IDT	46

83.5.1	Struttura della tabella IDT	46
83.5.2	Codice per la costruzione di una tabella IDT	48
83.5.3	Attivazione della tabella IDT	50
83.5.4	Lo stato della pila al verificarsi di un'interruzione	53
83.5.5	Bozza di un gestore di interruzioni	55
83.5.6	Una funzione banale per il controllo delle interruzioni 61	
83.5.7	Privilegi e protezioni	62
83.6	Gestione delle interruzioni	64
83.6.1	Eccezioni	64
83.6.2	PIC e rimappatura delle interruzioni	66
83.6.3	Procedura generalizzata per la gestione delle interruzioni	69
83.6.4	Attivazione	73
83.7	Gestione del temporizzatore: PIT, ovvero «programmable interval timer»	73
83.8	Tastiera PS/2	77
83.9	Gestione di dischi PATA	79
83.9.1	Modalità di accesso	81
83.9.2	Registri per la gestione delle unità ATA	83
83.9.3	Alcuni comandi	90
83.9.4	Individuazione dei bus utilizzati	98
83.9.5	Azzeramento dello stato dei dispositivi	99
83.9.6	Controllo delle interruzioni	100

83.9.7	Verifica dell'esito di un comando	100
83.9.8	Identificazione delle unità	101
83.9.9	Scomposizione dell'indirizzo	104
83.9.10	Lettura LBA28 PIO	106
83.9.11	Scrittura LBA28 PIO	110
83.10	PCI	112
83.10.1	Registri e porte	113
83.10.2	Strutture dei dati	114
83.10.3	Raccolta delle informazioni	127
83.11	NE2000	132
83.11.1	Memoria interna	132
83.11.2	Coda di ricezione	135
83.11.3	Tampone di trasmissione	138
83.11.4	Trasferimento dati tra la memoria interna e quella dell'elaboratore	138
83.11.5	Registri e porte	140
83.11.6	Procedura di riconoscimento	150
83.11.7	Procedura di inizializzazione	152
83.11.8	Trasmissione di un pacchetto	166
83.11.9	Ricezione	172
83.12	IPv4 in una rete Ethernet	182
83.12.1	ARP	183
83.12.2	IPv4	189
83.12.3	Il codice di controllo del TCP/IP	196

83.12.4	ICMP	199
83.12.5	UDP	205
83.12.6	TCP	211
83.13	Riferimenti	229

CLI 24 INB 22 IRET 53 LGDT 42 LIDT 50 OUTB 22 STI 24

Questo capitolo raccoglie le informazioni basilari per la realizzazione di un sistema autonomo, privo però di funzionalità utili. Per chiarire i concetti raccolti in questo capitolo è molto importante affiancare la lettura di *Intel Architectures Software Developer's Manual, System Programming Guide* (ottenibile presso <http://developer.intel.com/products/processor/manuals/index.htm>).

Per affrontare il capitolo è opportuno prendere prima confidenza con la sezione 65.5, nella quale si guida a realizzare un programma, avviato attraverso GRUB o SYSLINUX, in grado semplicemente di visualizzare un messaggio sullo schermo.

Per rendere agevoli gli esperimenti descritti in questo capitolo, è necessario utilizzare Bochs o QEMU, ovvero di un emulatore di architettura x86-32. Supponendo di avere predisposto un file-immagine di un dischetto, in cui si avvia il proprio kernel sperimentale attraverso GRUB 1 o di SYSLINUX, conviene predisporre uno script per l'avvio di Bochs o di QEMU senza doversi preoccupare di altro:

```
#!/bin/sh
bochs -q 'boot:a' 'floppya: 1_44=floppy.img, status=inserted'
```

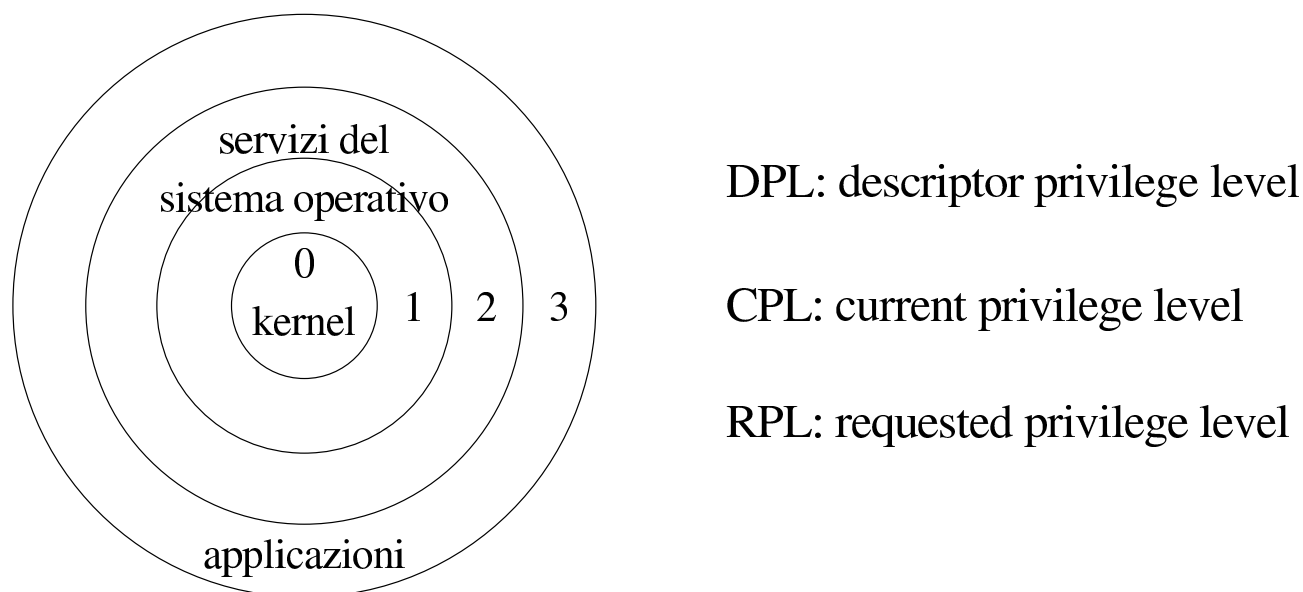
```
#!/bin/sh
qemu -fda floppy.img -boot a
```

Come si comprende intuitivamente, il file-immagine del dischetto deve chiamarsi `'floppy.img'`.

83.1 Privilegi dei segmenti

La gestione dei microprocessori x86-32 in modalità protetta, prevede che i dati e i processi elaborativi siano generalmente classificati in base a dei privilegi, secondo un modello ad anelli. «

Figura 83.3. Modello di rappresentazione dei privilegi ad anelli.



I microprocessori x86-32 definiscono precisamente quattro anelli, numerati da zero a tre e vi si attribuiscono convenzionalmente delle competenze: al livello zero, corrispondente all'anello centrale, competono i privilegi più importanti, ovvero quelli del kernel; al livello tre, corrispondente all'anello più esterno, competono i privilegi meno importanti, ovvero quelli delle applicazioni. In altri termini, gli anelli più interni, a cui corrisponde un valore numericamente minore, sono dati privilegi maggiori rispetto a quelli più esterni.

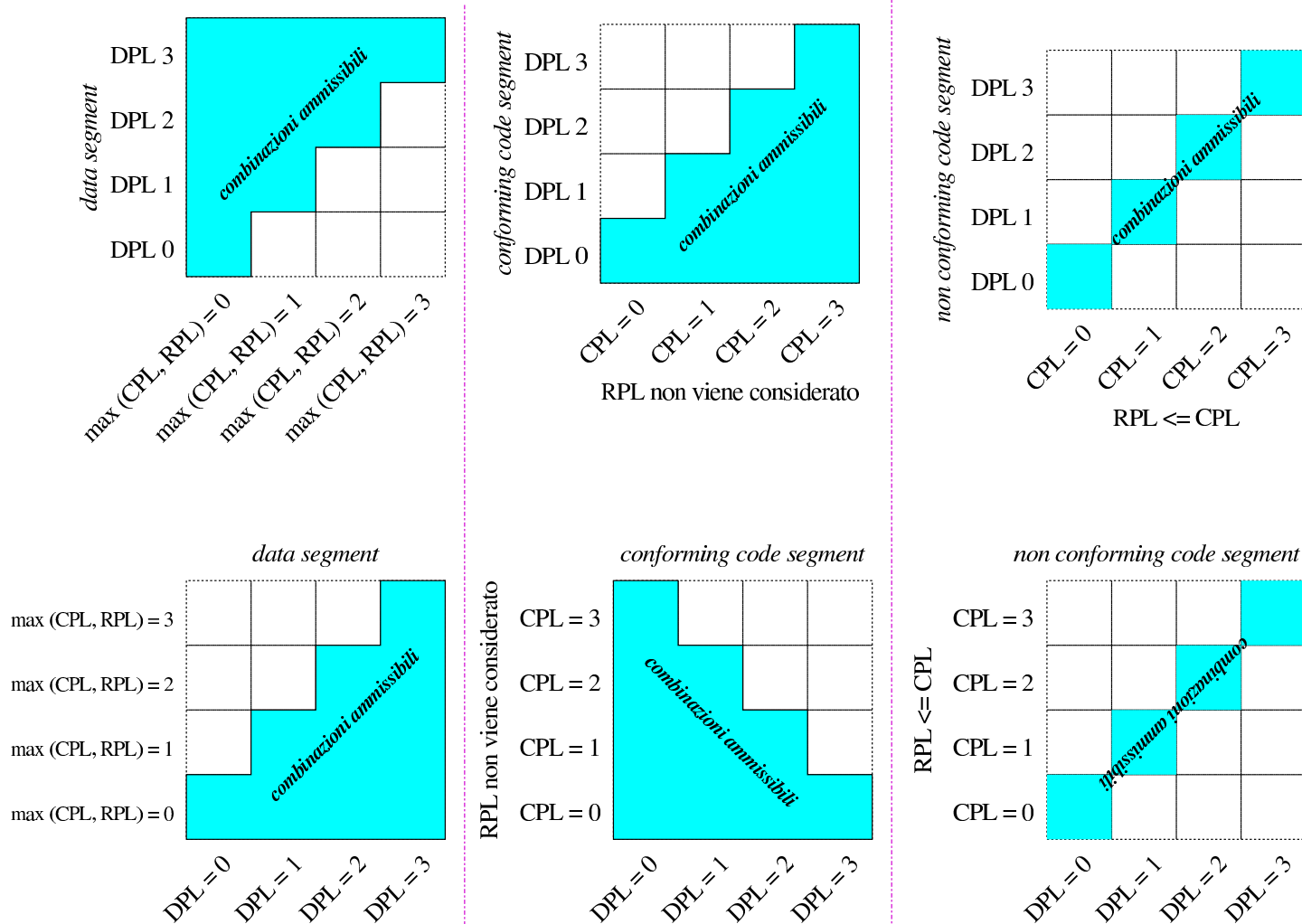
83.1.1 DPL, CPL e RPL

«

Nei microprocessori x86-32 si usano delle definizioni per rappresentare tre contesti diversi in cui sono considerati i privilegi ad anelli: DPL, ovvero *descriptor privilege level*; CPL, ovvero *current privilege level*; RPL, ovvero *requested privilege level*. La sigla DPL rappresenta un privilegio attribuito a un «oggetto»; pertanto, il descrittore del tale oggetto porta con sé l'indicazione del privilegio a cui questo fa riferimento. La sigla CPL rappresenta il privilegio attivo per il processo elaborativo in corso di esecuzione. La sigla RPL rappresenta il privilegio richiesto per accedere a un certo oggetto e potrebbe essere diverso dal privilegio del processo elaborativo attuale (CPL).

Le situazioni in cui si applica il controllo dei privilegi sono varie, ma semplificando in modo un po' approssimativo si presentano tre possibilità fondamentali: codice che deve raggiungere dati; codice che deve raggiungere altro codice di tipo «conforme»; codice che deve raggiungere altro codice di tipo «non conforme». L'aggettivo «conforme» associato al codice serve solo a distinguere due comportamenti alternativi e non ha molta importanza individuare il significato originale dato al termine usato.

Figura 83.4. Combinazioni tra CPL, RPL e DPL, nelle tre situazioni più comuni, secondo due prospettive alternative.



Codice che deve raggiungere dei dati

Quando si deve accedere a dei dati, in un'area di memoria a cui ci si riferisce attraverso un descrittore, il livello di privilegio di tale descrittore (DPL) deve essere numericamente maggiore, sia di CPL, sia di RPL. Pertanto il processo elaborativo ha accesso a dati meno importanti del proprio livello; se però si vuole limitare ulteriormente l'importanza dei dati a cui si può accedere, si può utilizzare un valore RPL numericamente più alto del proprio livello effettivo.

Codice che deve raggiungere altro codice conforme

Quando il codice in corso di esecuzione deve saltare verso un'altra posizione, qualificata come «conforme», il descrittore che si riferisce alla memoria che contiene tale nuovo codice deve avere un livello di privilegio numericamente minore o uguale a quello effettivo del codice di origine. Pertanto il processo elaborativo può spostarsi a utilizzare codice con lo stesso livello di privilegio o a codice con un privilegio più importante. In tal caso, il valore di RPL non viene considerato.

Per «codice conforme» vanno intese quindi delle procedure che sono «sicure» per tutti i processi con importanza inferiore o al massimo uguale a quella delle procedure stesse. La conformità si può riferire al concetto di standardizzazione delle procedure, come nel caso di librerie di funzioni.

Codice che deve raggiungere altro codice non conforme

Quando il codice in corso di esecuzione deve saltare verso un'altra posizione, qualificata come «non conforme», il descrittore che si riferisce alla memoria che contiene tale nuovo codice deve avere un livello di privilegio identico a quello effettivo del codice di origine. In questo caso, il valore di RPL è importante solo in quanto deve essere numericamente inferiore o uguale a quello di CPL.

Il codice «non conforme» è quello che non ha requisiti di standardizzazione e di sicurezza tali da consentire una condivisione con i processi elaborativi con un privilegio meno importante; d'altra parte, per motivi diversi, non è nemmeno abbastanza sicuro da poter essere riutilizzato da processi elaborativi più importanti.

83.1.2 Pila dei dati

Per ogni livello di privilegio che può assumere un processo elaborativo, deve essere disponibile una pila dei dati differente. Pertanto, il processo elaborativo che sta funzionando con un livello di privilegio attuale (CPL) pari a zero, deve utilizzare una pila che si colloca in un'area di memoria qualificata da un livello di privilegio del descrittore (DPL) pari a zero. Lo stesso vale per gli altri livelli di privilegio. Ciò che qui non viene spiegato è il modo in cui un processo può modificare il proprio livello di privilegio attuale (CPL) e acquisire, di conseguenza, un'altra pila dei dati.

Quando il processo elaborativo raggiunge del codice «conforme» a partire da un livello di privilegio attuale meno importante di quello del codice in questione, il valore di CPL non cambia, quindi non cambia nemmeno la pila dei dati relativa al processo elaborativo.

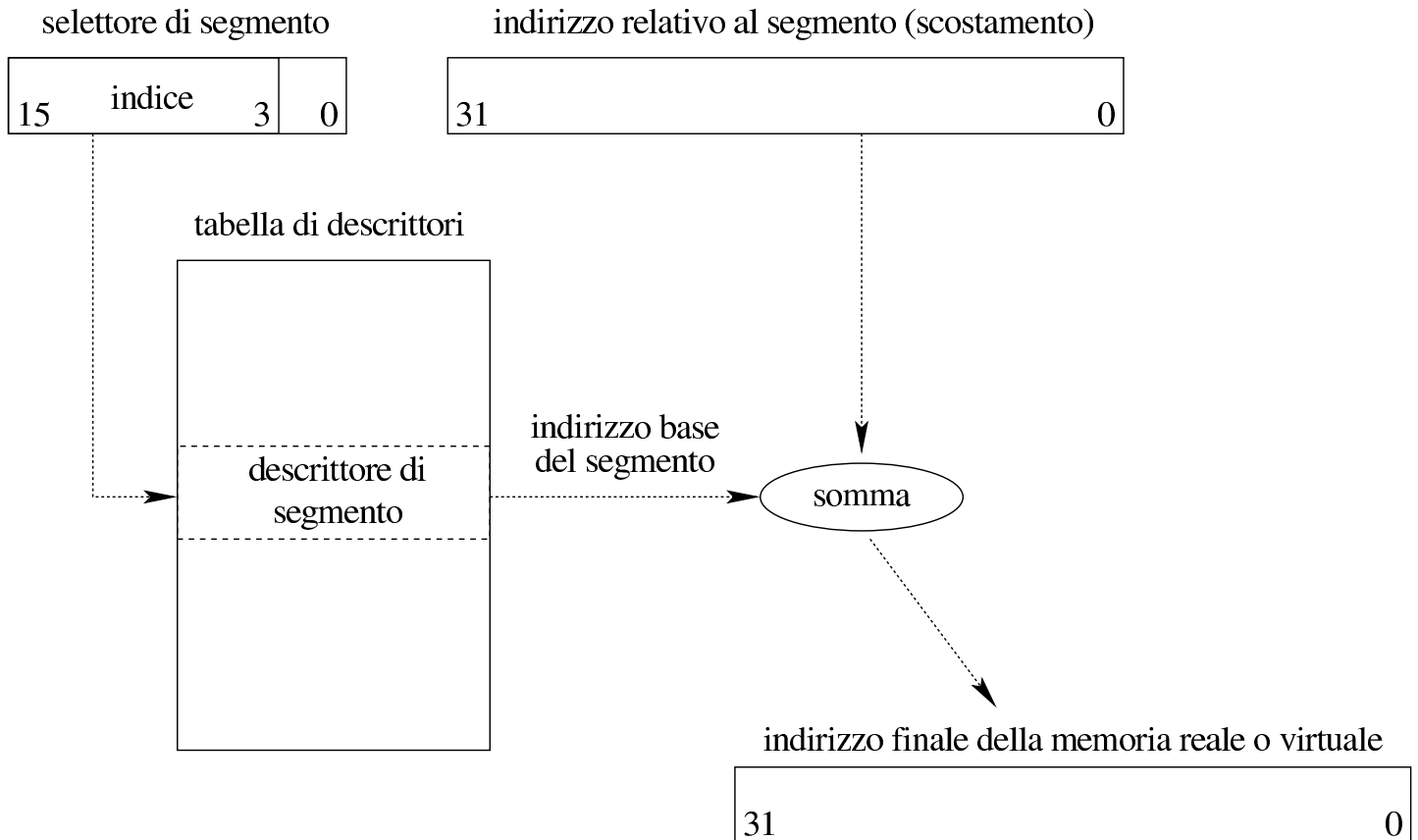
83.1.3 Segmenti, selettori e registri

La gestione della memoria di un microprocessore x86-32, funzionante in modalità protetta, richiede che la memoria sia organizzata in segmenti, i quali, eventualmente possono essere suddivisi in pagine di memoria virtuale. A ogni modo, i segmenti rappresentano sempre il punto di riferimento principale e vengono specificati attraverso l'aiuto di registri di segmento.

Per individuare un indirizzo di memoria (reale o virtuale), si parte da un *selettore*, contenuto in un registro di segmento appropriato al contesto, dal quale si ottiene un indice per selezionare una voce da una tabella di descrittori. Attraverso l'indice si individua il descrittore di un segmento, del quale si ottiene l'indirizzo iniziale nella memoria (reale o virtuale). A questo indirizzo iniziale va poi aggiunto

uno scostamento che rappresenta l'indirizzo relativo all'interno del segmento.

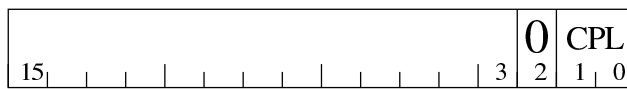
Figura 83.5. Determinazione dell'indirizzo di memoria, attraverso l'indicazione di un indirizzo relativo a un segmento.



I registri all'interno dei quali vanno inseriti i selettori di segmento possono essere *CS* (*code segment*), *SS* (*stack segment*), *DS* (*data segment*), *ES*, *FS* e *GS* (sono tutti registri a 16 bit). In particolare, il registro *CS* serve a individuare il segmento in cui è in corso di esecuzione il codice attuale; il registro *SS* individua il segmento in cui si trova la pila dei dati utilizzata dal processo elaborativo attuale; il registro *DS* e gli altri individuano dei segmenti contenenti altri tipi di dati, a cui il processo elaborativo in corso deve accedere. Il valore che si scrive in questi registri è il selettore di segmento, il quale va interpretato secondo lo schema della figura successiva.

Figura 83.7. Interpretazione dello stato dei registri *CS* e *SS*.

*contenuto effettivo
dei registri CS e SS*



CPL: livello di privilegio effettivo

indice del descrittore del segmento utilizzato

Per quanto riguarda specificatamente i segmenti, i privilegi individuati dalla sigla DPL sono quelli annotati nel descrittore di segmento (della tabella relativa) al quale si vuole accedere. Esistono comunque altri contesti in cui compaiono dei privilegi di oggetti a cui si fa riferimento con un descrittore.

83.2 Funzioni di utilità generale

« Nella realizzazione di un sistema indipendente, per architettura x86-32, sono necessarie delle piccole funzioni, attraverso le quali si richiamano delle istruzioni in linguaggio assembler. Ciò che viene usato o che può essere usato nelle sezioni successive, viene riassunto qui.

83.2.1 Porte I/O

« Per comunicare con i dispositivi è necessario poter leggere e scrivere attraverso delle porte di comunicazione interne. Per fare questo si usano frequentemente le istruzioni **'INB'** e **'OUTB'** del linguaggio assembler. Quelli che seguono sono i listati di due funzioni con lo stesso nome, per consentire di usare queste istruzioni attraverso il linguaggio C:

```
.globl inb
#
inb:
    enter $4, $0
    pusha
    .equ inb_port, 8           # Primo parametro.
    .equ inb_data, -4        # Variabile locale.
    mov inb_port(%ebp), %edx  # Successivamente viene usato
                             # solo DX.
    inb %dx, %al
    mov %eax, inb_data(%ebp) # Salva EAX nella variabile
                             # locale.
    popa
    mov inb_data(%ebp), %eax # Recupera EAX che rappresenta
                             # il valore restituito dalla
                             # funzione.
    leave
    ret
```

```
.globl outb
#
outb:
    enter $0, $0
    pusha
    .equ outb_port, 8        # Primo parametro.
    .equ outb_data, 12      # Secondo parametro.
    mov outb_port(%ebp), %edx # Successivamente viene usato
                             # solo DX.
    mov outb_data(%ebp), %eax # Successivamente viene usato
                             # solo AL.
    outb %al, %dx
    popa
    leave
    ret
```

I prototipi delle due funzioni, da usare nel linguaggio C sono i

seguenti:

```
unsigned int inb (unsigned int port);
```

```
void outb (unsigned int port, unsigned int data);
```

Il significato della sintassi è molto semplice: la funzione *inb()* riceve come argomento il numero di una porta e restituisce il valore, costituito da un solo byte, che da quella si può leggere; la funzione *outb()* riceve come argomenti il numero di una porta e il valore, rappresentato sempre solo da un byte, che a quella porta va scritto, senza restituire alcunché.

Nei prototipi si usano interi normali, invece di byte, ma poi viene considerata solo la porzione del byte meno significativo.

83.2.2 Sospensione e ripristino delle interruzioni hardware

«

Attraverso le istruzioni ‘**CLI**’ e ‘**STI**’ è possibile, rispettivamente, sospendere il riconoscimento delle interruzioni hardware (IRQ) e ripristinarlo. Le due funzioni seguenti si limitano a tradurre queste due istruzioni in funzioni utilizzabili con il linguaggio C:

```
.globl cli
#
cli:
    cli
    ret
```



```
.globl sti
#
sti:
    sti
    ret
```

Evidentemente, i prototipi per il linguaggio C sono semplicemente così:

```
void cli (void);
```

```
void sti (void);
```

83.3 Utilizzo dello schermo VGA a caratteri

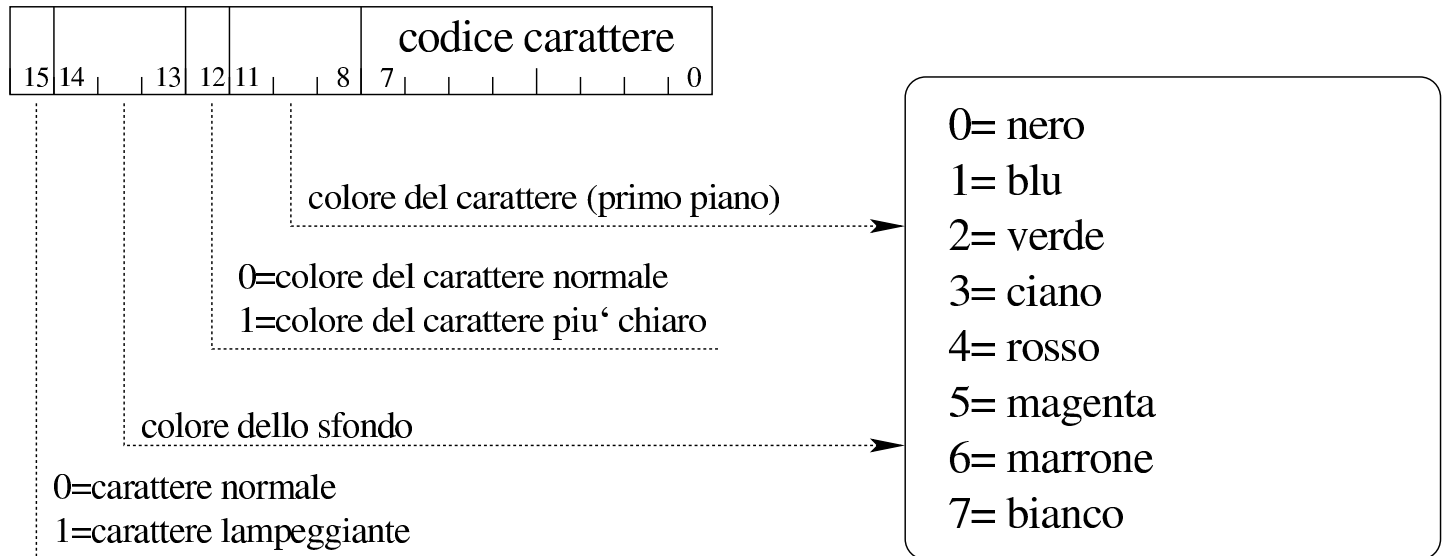
Per poter scrivere un programma che utilizzi autonomamente le risorse hardware, senza avvalersi di un sistema operativo, la prima cosa di cui ci si deve prendere cura è la visualizzazione di messaggi sullo schermo. Di norma si parte dal presupposto che un elaboratore x86-32 disponga di uno schermo controllato da un adattatore VGA, sul quale è possibile visualizzare del testo puro e semplice, senza dover affrontare troppe complicazioni.

83.3.1 Memoria dello schermo a caratteri

Per visualizzare un messaggio su uno schermo VGA, quando non è possibile usare le funzioni del BIOS, perché si sta lavorando in modalità protetta, è necessario scrivere in una porzione di memoria che parte dall'indirizzo $B8000_{16}$, utilizzando una sequenza a 16 bit, dove gli otto bit più significativi costituiscono un codice che descrive

i colori da usare per il carattere e il suo sfondo, mentre il secondo contiene il carattere da visualizzare.

Figura 83.12. Organizzazione dei 16 bit con i quali si rappresenta un carattere sullo schermo.



La figura mostra come va costruito il carattere da visualizzare sullo schermo. Per esempio, un colore indicato come 28_{16} genera un testo di colore bianco, a intensità normale, su sfondo verde, mentre $A0_{16}$ genera un testo lampeggiante nero su sfondo verde.

83.3.2 Cursore

«

Per visualizzare del testo sullo schermo, è sufficiente assemblare i caratteri nel modo descritto sopra, collocandoli in memoria a partire dall'indirizzo $B8000_{16}$, sapendo che presumibilmente lo schermo è organizzato a righe da 80 colonne (pertanto ogni riga utilizza 160 byte e una schermata normale da 25 righe occupa complessivamente 4000 byte). Ma la visualizzazione del testo è indipendente dalla gestione del cursore e per collocarlo da qualche parte sullo schermo, occorre comunicare con l'adattatore VGA attraverso dei registri specifici.

Prima di comunicare con l'adattatore VGA per collocare il cursore, occorre definire le coordinate del cursore. Per questo occorre contare i caratteri, contando da zero. Per esempio, ammesso di voler collocare il cursore in corrispondenza della seconda colonna della ventesima riga, su uno schermo da 80 colonne per 25 righe, la posizione che si vuole raggiungere è $19 \times 80 + 2 - 1 = 1521$. Questo numero corrisponde a $05F1_{16}$.

Con l'ausilio della funzione *outb()* descritta in un altro capitolo, si comunica con l'adattatore VGA la posizione del cursore nel modo seguente:

```
...
    outb (0x3D4, 14);    // Prima parte.
    outb (0x3D5, 0x05);

    outb (0x3D4, 15);    // Seconda parte.
    outb (0x3D5, 0xF1);
...
```

Come si vede, l'indirizzo del cursore va dato in due fasi, dividendolo in due byte.

83.4 GDT

Nei microprocessori x86-32, per poter accedere alla memoria quando si sta operando in modalità protetta,¹ è indispensabile dichiarare la mappa dei segmenti di memoria attraverso una o più tabelle di descrizione. Tra queste è indispensabile la dichiarazione della tabella GDT, ovvero *global description table*, collocata nella stessa memoria centrale.²

La tabella GDT deve essere predisposta dal sistema operativo, prima di ogni altra cosa; di norma ciò avviene prima di far passare il micro-

processore in modalità protetta, in quanto tale passaggio richiede che la tabella sia già presente per consentire al microprocessore di conoscere i permessi di accesso. Tuttavia, se si utilizza un programma per l'avvio del sistema operativo, si potrebbe trovare il microprocessore già in modalità protetta, con una tabella GDT provvisoria, predisposta in modo tale da consentire l'accesso alla memoria senza limitazioni e in modo lineare.³ Per esempio, questo è ciò che avviene con un sistema di avvio aderente alle specifiche *multiboot*, come nel caso di GRUB 1. Ma anche così, il sistema operativo deve comunque predisporre la propria tabella GDT, rimpiazzando la precedente.

Negli esempi che appaiono nelle sezioni successive, si fa riferimento alla predisposizione di una tabella GDT, a partire da un sistema che è già in modalità protetta, essendo consentito di accedere a tutta la memoria, linearmente, senza alcuna limitazione.

83.4.1 Privilegi

«

Negli esempi che vengono mostrati, i privilegi corrispondono sempre all'anello zero, onde evitare qualunque tipo di complicazione. Tuttavia, è evidente che un sistema operativo comune deve invece gestire in modo più consapevole questo problema.

83.4.2 Organizzazione e contenuti della tabella GDT

«

Inizialmente conviene considerare la tabella GDT in modo semplificato, organizzata a righe e colonne, come si vede nello schema successivo, dal momento che nella realtà i dati sono spezzettati e sparpagliati nello spazio a disposizione. Le righe di questa tabella possono essere al massimo 8192 e ogni riga costituisce il *descrittore* di un

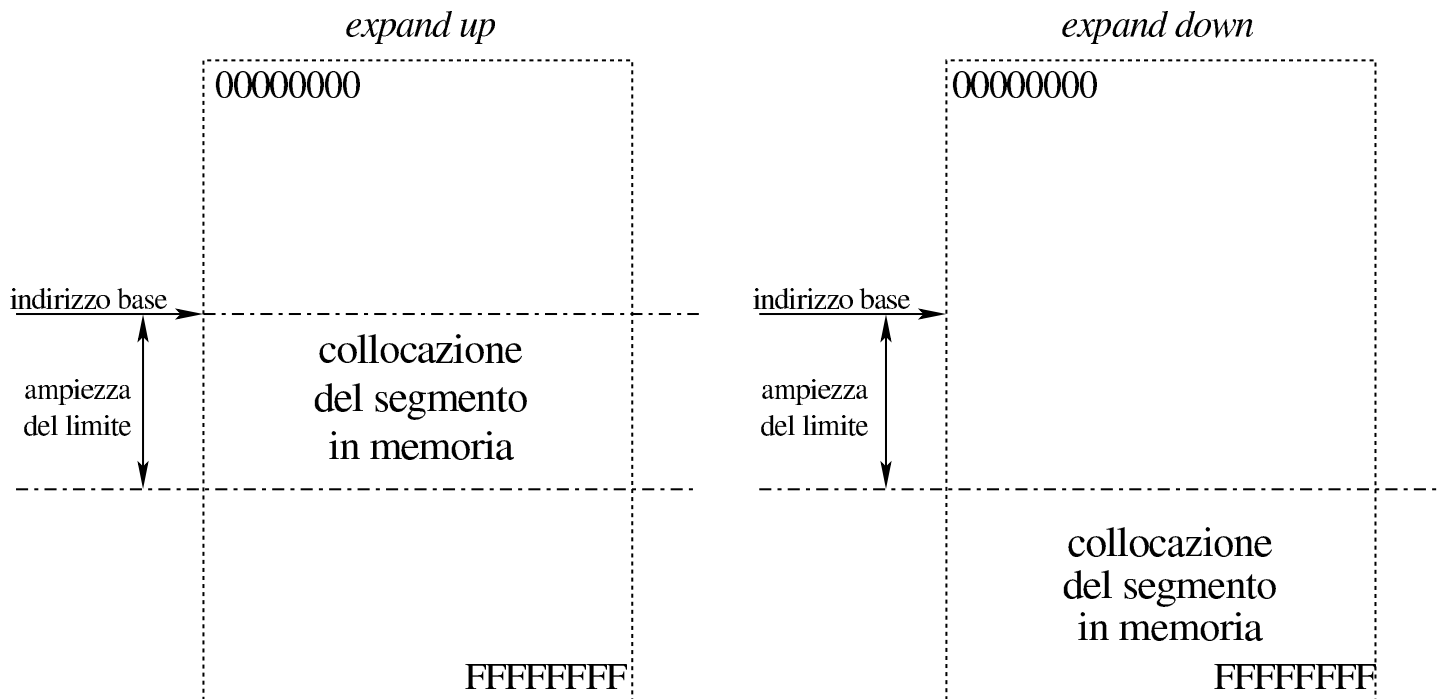
segmento di memoria, il quale, eventualmente, può sovrapporsi ad altre aree.⁴

	base	limite	attributi
<i>descrittore 0</i>	32 bit	20 bit	12 bit
<i>descrittore 1</i>	32 bit	20 bit	12 bit
<i>descrittore 2</i>	32 bit	20 bit	12 bit
...			
<i>descrittore n</i>	32 bit	20 bit	12 bit

Come si vede, gli elementi dominanti delle voci che costituiscono la tabella, ovvero dei descrittori di segmento, sono la «base» e il «limite». Il primo rappresenta l'indirizzo iniziale del segmento di memoria a cui si riferisce il descrittore; il secondo rappresenta in linea di massima l'estensione di questo segmento.

Il modo corretto di interpretare il valore che rappresenta il limite dipende da due attributi: la granularità e la direzione di espansione. Come si può vedere il valore attribuibile al limite è condizionato dalla disponibilità di soli 20 bit, con i quali si può rappresentare al massimo il valore $FFFF_{16}$, pari a 1048575_{10} . L'attributo di granularità consente di specificare se il valore del limite riguarda byte singoli o se rappresenta multipli di 4096 byte (ovvero 1000_{16} byte). Evidentemente, con una granularità da 4096 byte è possibile rappresentare valori da 00000000_{16} a $FFFFF000_{16}$.

Figura 83.15. Confronto tra un limite da interpretare in modalità normale (a sinistra), rispetto a un limite da interpretare secondo un attributo di espansione in basso.

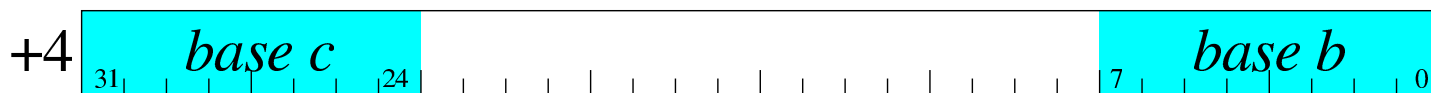
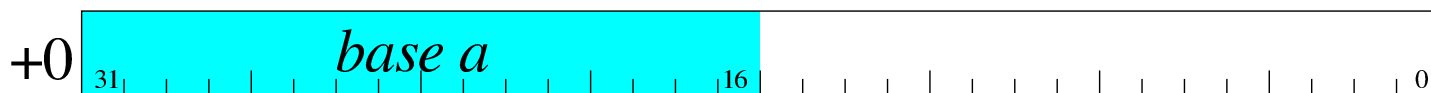


La direzione di espansione serve a determinare come si colloca l'area del segmento; si distinguono due casi: da *base*, fino a $base + (\limite \times \text{granularità})$ incluso; oppure da $base + (\limite \times \text{granularità}) + 1$ a $FFFFFFF_{16}$. Il concetto è illustrato dalla figura già apparsa.

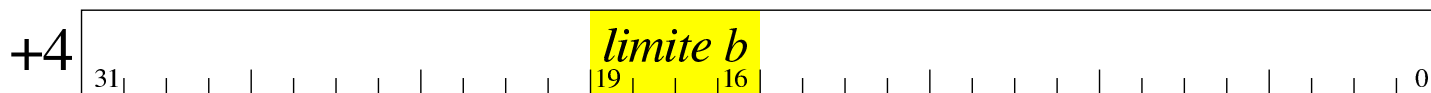
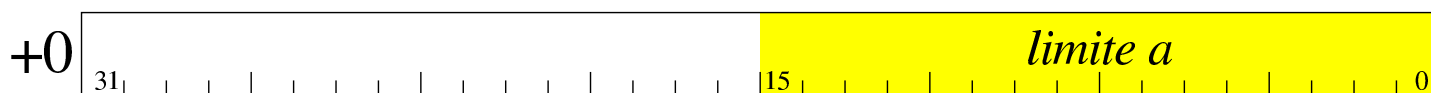
83.4.3 Struttura effettiva della tabella GDT

«

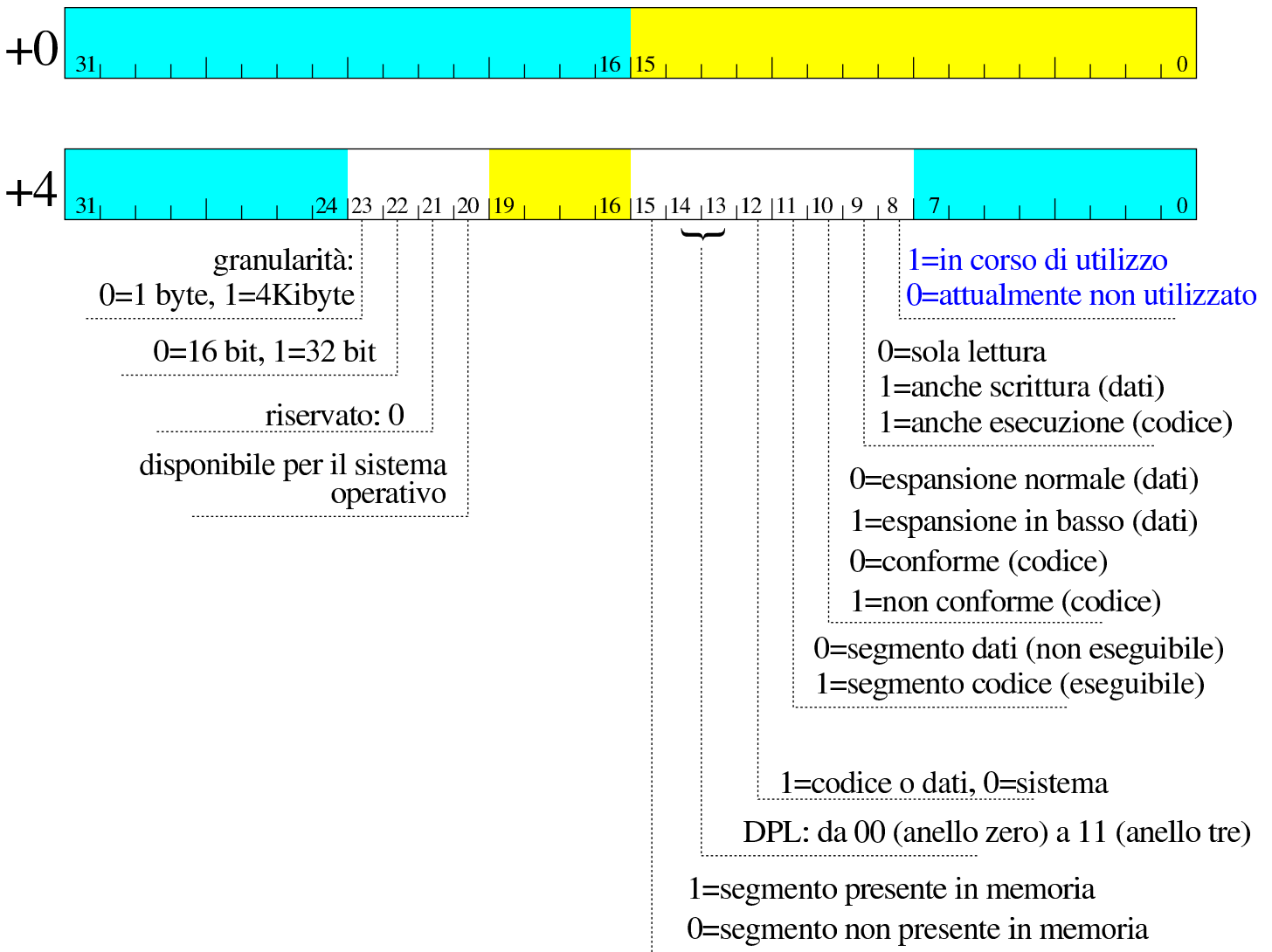
Nella realtà, la tabella GDT è formata da un array di descrittori, ognuno dei quali è composto da 8 byte, rappresentati qui in due blocchi da 32 bit, come nello schema successivo, dove viene evidenziata la porzione che riguarda l'indicazione dell'indirizzo iniziale del segmento di memoria:



L'indirizzo iniziale del segmento di memoria va ricomposto, utilizzando i bit 16-31 del primo blocco a 32 bit; quindi aggiungendo a sinistra i bit 0-7 del secondo blocco a 32 bit; infine aggiungendo a sinistra i bit 24-31 del secondo blocco a 32 bit. Anche il valore del limite del segmento di memoria risulta frammentato:



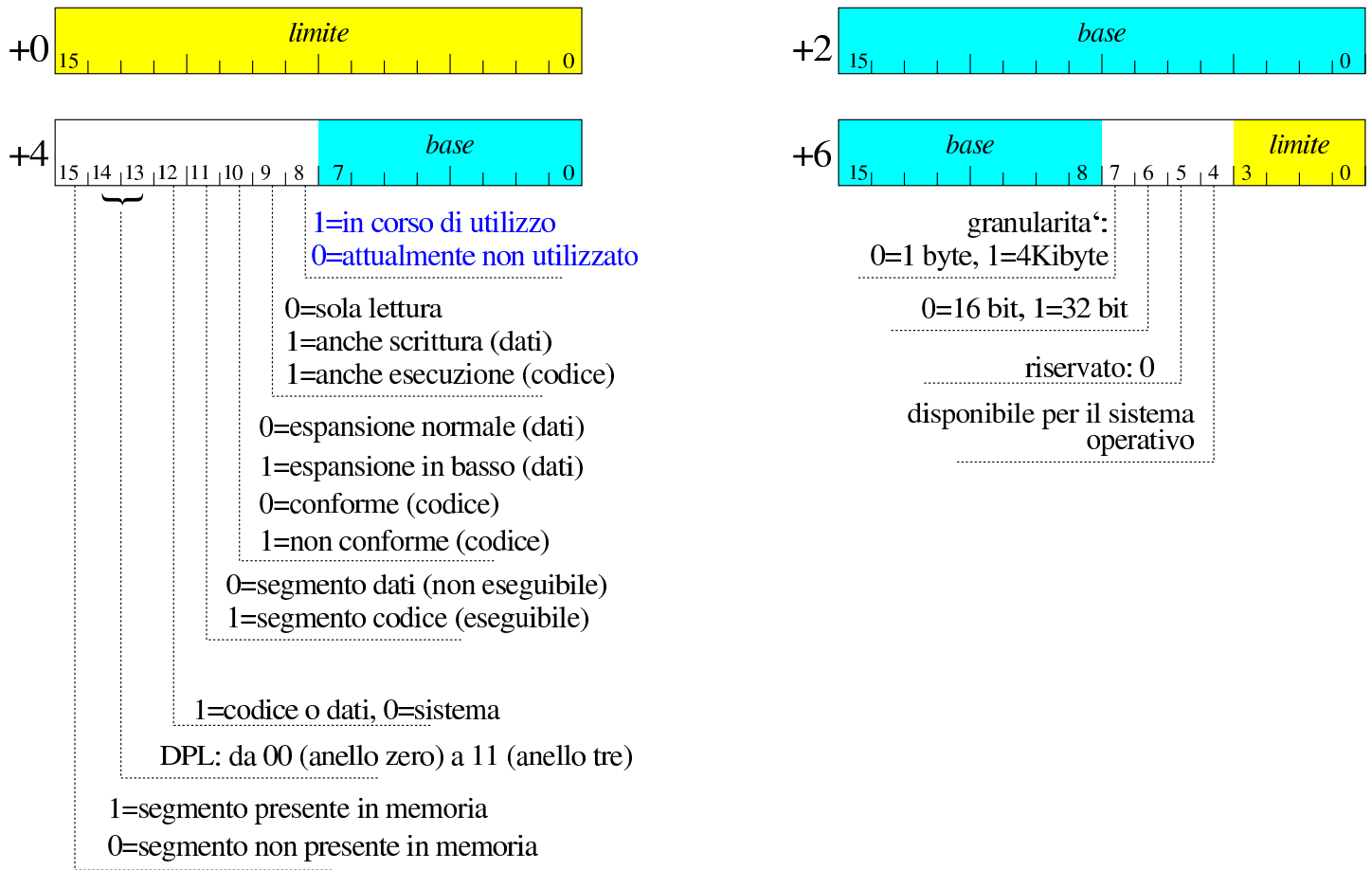
Il limite del segmento di memoria va ricomposto, utilizzando i bit 0-15 del primo blocco a 32 bit, aggiungendo a sinistra i bit 16-19 del secondo blocco a 32 bit. Nel disegno successivo si illustrano gli altri attributi, considerando che si tratti di un descrittore di memoria per codice o dati; in altri termini, il bit 12 (il tredicesimo) del secondo blocco a 32 bit **deve essere impostato a uno**:



A proposito del bit che rappresenta il tipo di espansione o la conformità, in generale va usato con il valore zero, a indicare che il limite rappresenta l'espansione del segmento, a partire dalla sua origine, oppure che l'interpretazione del codice è da intendere in modo «conforme» per ciò che riguarda i privilegi. Il bit di accesso in corso (il bit numero 8, nel secondo blocco da 32 bit) viene aggiornato dal microprocessore, ma normalmente solo se all'inizio appare azzerato.

Va ricordato che i microprocessori x86-32 scambiano l'ordine dei byte in memoria. Pertanto, gli schemi mostrati sono validi solo se l'accesso alla memoria avviene a blocchi da 32 bit, perché diversamente occorrerebbe tenere conto di tali scambi. Per questa stessa ragione, il descrittore di un segmento di memoria è stato mostrato diviso in due blocchi da 32 bit, invece che in uno solo da 64, dato che l'accesso non può avvenire simultaneamente per modificare o leggere un descrittore intero.

Quando si deve predisporre una tabella GDT prima di essere passati al funzionamento in modalità protetta, ovvero quando non ci si può avvalere di un sistema di avvio che offre una modalità protetta provvisoria, occorre ragionare a blocchi da 16 bit, non essendoci la possibilità di usare istruzioni a 32. Pertanto, ognuno dei blocchi descritti va invertito, come si può vedere nel disegno successivo:

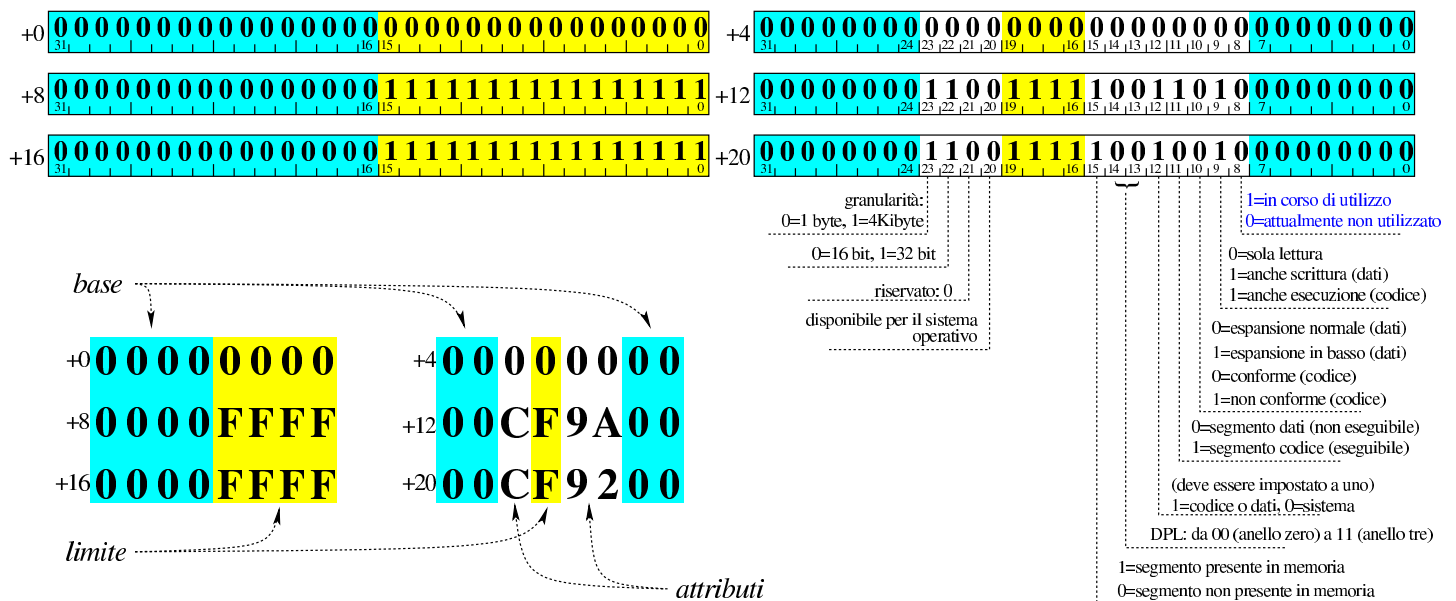


83.4.4 Tabella GDT elementare

«

Una tabella GDT elementare, con la quale si voglia dichiarare tutta la memoria centrale (al massimo fino a 4 Gibyte), in modo lineare e senza distinzione di privilegi tra il codice e i dati, richiede almeno tre descrittori: un descrittore nullo iniziale, obbligatorio; un descrittore per il segmento codice che si estende su tutta la superficie della memoria; un altro descrittore identico, ma riferito ai dati. In pratica, a parte il descrittore nullo iniziale, servono almeno due descrittori, uno per il codice e l'altro per i dati, sovrapposti, entrambi attribuiti all'anello zero (quello principale). Questa è di norma la situazione che viene proposta negli esempi in cui si dimostra il funzionamento di un kernel elementare; nel disegno della figura 83.20 si può vedere sia in binario, sia in esadecimale.

Figura 83.20. Esempio di tabella GDT elementare.



83.4.5 Costruzione di una tabella GDT

Per costruire una tabella GDT è complicato usare una struttura per tentare di riprodurre la suddivisione degli elementi di un descrittore di segmento; pertanto, qui viene proposta una soluzione con una suddivisione che si riduce a due blocchi da 32 bit:

```
#include <stdint.h>
typedef struct {
    uint32_t w0;
    uint32_t w1;
} gdt_descriptor_t;
```

La funzione successiva riceve come argomento un array di descrittori di segmento, con l'indicazione dell'indice a cui si vuole fare riferimento e degli attributi che gli si vogliono associare. Va però osservato che i nomi dei parametri *access* e *granularity* rappresentano una semplificazione, nel senso che *access* si riferisce agli attributi che vanno dal segmento presente in memoria fino al segmento in

corso di utilizzo, mentre *granularity* va dalla granularità fino ai bit che rappresentano un segmento riservato e disponibile:

```
#include <stdint.h>
static void
gdt_descriptor_set (gdt_descriptor_t *gdt,
                   int                descr,
                   uint32_t           base,
                   uint32_t           limit,
                   uint32_t           access,
                   uint32_t           granularity)
{
    //
    // Azzera la voce selezionata.
    //
    gdt[descr].w0 = 0;
    gdt[descr].w1 = 0;
    //
    // Trasferisce l'ampiezza del segmento (limit).
    //
    gdt[descr].w0 = gdt[descr].w0 | (limit & 0x0000FFFF);
    gdt[descr].w1 = gdt[descr].w1 | (limit & 0x000F0000);
    //
    // Trasferisce l'indirizzo iniziale del segmento (base).
    //
    gdt[descr].w0
    = gdt[descr].w0 | ((base << 16) & 0xFFFF0000);
    gdt[descr].w1
    = gdt[descr].w1 | ((base >> 16) & 0x000000FF);
    gdt[descr].w1
    = gdt[descr].w1 | (          base & 0xFF000000);
    //
    // Trasferisce gli attributi di accesso e altri attributi
    // vicini.
    //
}
```

```

gdt[descr].w1
= gdt[descr].w1 | ((access << 8) & 0x0000FF00);
//
// Trasferisce la granularità e altri attributi vicini.
//
gdt[descr].w1
= gdt[descr].w1 | ((granularity << 20) & 0x00F00000);
}

```

Per usare questa funzione occorre prima dichiarare l'array di descrittori di segmento. L'esempio seguente serve a riprodurre la tabella elementare della figura 83.20:

```

...
static gdt_descriptor_t gdt[3]; // Tabella GDT con tre voci.
...
gdt_descriptor_set (gdt, 0, 0, 0, 0, 0); // [1]
gdt_descriptor_set (gdt, 1, 0, 0xFFFFFFFF, 0x9A, 0xC); // [2]
gdt_descriptor_set (gdt, 2, 0, 0xFFFFFFFF, 0x92, 0xC); // [3]
// [1] Obbligatorio.
// [2] Codice conforme.
// [3] Dati.
...

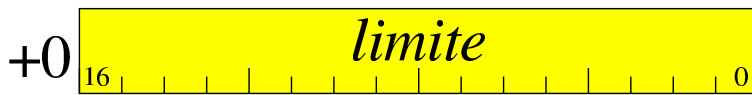
```

Nell'esempio, l'array *gdt[]* viene creato specificando l'uso di memoria «statica», nell'ipotesi che ciò avvenga dentro una funzione; diversamente, può trattarsi di una variabile globale senza vincoli particolari.

83.4.6 Attivazione della tabella GDT

«

La tabella GDT può essere collocata in memoria dove si vuole (o dove si può), ma perché il microprocessore la prenda in considerazione, occorre utilizzare un'istruzione specifica con la quale si carica il registro **GDTR** (*GDT register*) a 48 bit. Questo registro non è visibile e si carica con l'istruzione '**LGDT**', la quale richiede l'indicazione dell'indirizzo di memoria dove si articola una struttura contenente le informazioni necessarie. Si tratta precisamente di quanto si vede nel disegno successivo:



In pratica, vengono usati i primi 16 bit per specificare la grandezza complessiva della tabella GDT e altri 32 bit per indicare l'indirizzo in cui inizia la tabella stessa. Tale indirizzo, sommato al valore specificato nel primo campo, deve dare l'indirizzo dell'ultimo byte della tabella stessa.

Dal momento che la dimensione di un descrittore della tabella GDT è di 8 byte, il valore del limite corrisponde sempre a $8 \times n - 1$, dove n è la quantità di descrittori della tabella. Così facendo, si può osservare che gli ultimi tre bit del limite sono sempre impostati a uno.

Nel disegno è stato mostrato chiaramente che il primo campo da 16 bit va considerato in modo separato. Infatti, si intende che l'accesso in lettura o in scrittura vada fatto lì esattamente a 16 bit, per-

ché diversamente i dati risulterebbero organizzati in un altro modo. Pertanto, nel disegno viene chiarito che il campo contenente l'indirizzo della tabella, inizia esattamente dopo due byte. In questo caso, con l'aiuto del linguaggio C è facile dichiarare una struttura che riproduce esattamente ciò che serve per identificare una tabella GDT:

```
#include <stdint.h>
typedef struct {
    uint16_t  limit;
    uint32_t  base;
} __attribute__ ((packed)) gdtr_t;
```

L'esempio mostrato si riferisce all'uso del compilatore GNU C, con il quale è necessario specificare l'attributo *packed*, per fare in modo che i vari componenti risultino abbinati senza spazi ulteriori di allineamento. Fortunatamente, il compilatore GNU C fa anche la cosa giusta per quanto riguarda l'accesso alla porzione di memoria a cui si riferisce la struttura.

Avendo definito la struttura, si può creare una variabile che la utilizza, tenendo conto che è sufficiente rimanga in essere solo fino a quando viene acquisita la tabella GDT relativa dal microprocessore:

```
...
    gdtr_t gdtr;
...
```

Per calcolare il valore che rappresenta la dimensione della tabella (il limite), occorre moltiplicare la dimensione di ogni voce (8 byte) per la quantità di voci, sottraendo dal risultato una unità. L'esempio presuppone che si tratti di tre voci in tutto:

```
...
    gdtr.limit = ((sizeof (gdt_descriptor_t)) * 3) - 1;
...
```

L'indirizzo in cui si trova la tabella GDT, può essere assegnato in modo intuitivo:

```
...
    gdtr.base  = (uint32_t) &gdt[0];
...
```

83.4.7 Verifica della tabella GDT

«

Le prime volte che si fanno esperimenti per ottenere l'attivazione di una tabella GDT, sarebbe il caso di verificare il contenuto di questa, prima di chiedere al microprocessore di attivarla. Infatti, un piccolo errore nel contenuto della tabella o in quello della struttura che contiene le sue coordinate, comporta generalmente un errore irreversibile. D'altra parte, proprio la complessità dell'articolazione delle voci nella tabella rende frequente il verificarsi di errori, anche multipli.

Ammesso di poter lavorare in una condizione tale da poter visualizzare qualcosa con una funzione *printf()*, la funzione seguente consente di vedere il contenuto di una tabella GDT, partendo dall'indirizzo della struttura che rappresenta il registro **GDTR** da caricare, ovvero dallo stesso indirizzo che dovrebbe ricevere il microprocessore, con l'istruzione '**LGDT**':

```
#include <stdint.h>
#include <inttypes.h>
#include <stdio.h>
static void
gdt_show (gdtr_t *gdtr)
{
```



```
gdt_descriptor_t *gdt = (gdt_descriptor_t *) gdtr->base;
int entries = (gdtr->limit + 1)
               / (sizeof (gdt_descriptor_t));
int  descr;
//
uint32_t base;
uint32_t limit;
uint32_t access;
uint32_t granularity;
//
printf ("gdt base: 0x%08X  limit: 0x%04X\n",
        gdtr->base, gdtr->limit);
//
printf ("          base          limit          "
        "access granularity\n");
//
for (descr = 0; descr < entries; descr++)
{
    base = limit = access = granularity = 0;
    //
    // Indirizzo del segmento di memoria.
    //
    base = base | ((gdt[descr].w0 >> 16) & 0x0000FFFF);
    base = base | ((gdt[descr].w1 << 16) & 0x00FF0000);
    base = base | ((gdt[descr].w1          ) & 0xFF000000);
    //
    // Estensione del segmento di memoria.
    //
    limit = limit | (gdt[descr].w0 & 0x0000FFFF);
    limit = limit | (gdt[descr].w1 & 0x000F0000);
    //
    // Attributi di accesso e di tipo.
    //
    access = access | ((gdt[descr].w1 >> 8) & 0x000000FF);
```

```

//
// Attributi di granularità.
//
granularity = granularity
                | ((gdt[descr].w1 >> 20) & 0x0000000F);

//
// Visualizza la voce della tabella.
//
printf ("gdt[%i] 0x%08" PRIX32 " 0x%06" PRIX32
        " 0x%04" PRIX32 " 0x%04" PRIX32 "\n",
        descr, base, limit, access, granularity);
}
}

```

Stando agli esempi già fatti, si dovrebbe vedere una cosa simile al testo seguente:

```

gdt base: 0x00106044  limit: 0x0017
      base      limit      access granularity
gdt[0] 0x00000000 0x000000 0x0000 0x0000
gdt[1] 0x00000000 0x0FFFFFFF 0x009A 0x000C
gdt[2] 0x00000000 0x0FFFFFFF 0x0092 0x000C

```

Il valore 17_{16} corrisponde a 23_{10} , pertanto, in questo caso, la tabella inizia all'indirizzo 00106044_{16} e termina all'indirizzo $0010605B_{16}$ compreso; inoltre la tabella occupa complessivamente 24 byte.

83.4.8 Istruzioni per l'attivazione

«

Per rendere operativo il contenuto della tabella GDT, va indicato al microprocessore l'indirizzo della struttura che contiene le coordinate della tabella stessa, attraverso l'istruzione '**LGDT**' (*load GDT*). Negli esempi seguenti si utilizzano istruzioni del linguaggio assembleatore,

secondo la sintassi di GNU AS; in quello seguente, in particolare, si suppone che il registro *EAX* contenga l'indirizzo in questione:

```
...  
    lgdt  (%eax)  # EAX contiene l'indirizzo della struttura.  
...
```

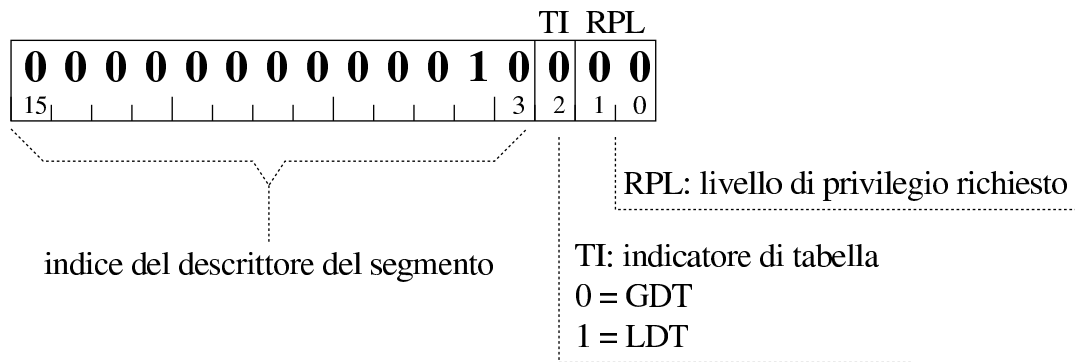
A questo punto, la tabella non viene ancora utilizzata dal microprocessore e occorre sistemare il valore di alcuni registri:

```
...  
    mov  $0x10, %eax  # Selettore di segmento.  
    mov  %eax, %ds  
    mov  %eax, %es  
    mov  %eax, %fs  
    mov  %eax, %gs  
    mov  %eax, %ss  
...
```

I registri in cui si deve intervenire sono *DS*, *ES*, *FS*, *GS* e *SS*, ma per assegnare loro un valore, occorre passare per la mediazione di un altro registro che in questo caso è *AX*. Il registro *DS* (*data segment*) e poi tutti gli altri citati, devono avere un selettore di segmento che punti al descrittore del segmento dati attuale, con la richiesta di privilegi adeguati e la specificazione che trattasi di un riferimento a una tabella GDT. Il disegno della figura successiva mostra come va interpretato il valore dell'esempio.

Figura 83.33. Selettore del segmento dati che riguarda sia *DS* con gli altri registri affini per l'accesso ai dati, sia *SS*, per la gestione della pila dei dati.

DS, ES, FS, GS, SS



Come si può vedere nel disegno, il valore 10_{16} assegnato ai registri destinati ai segmenti di dati, contiene l'indice 2_{10} per la tabella GDT, con la richiesta di privilegi pari a zero (ovvero il valore più importante). Il descrittore con indice due della tabella GDT è esattamente quello che è stato predisposto per i dati (figura 83.20).

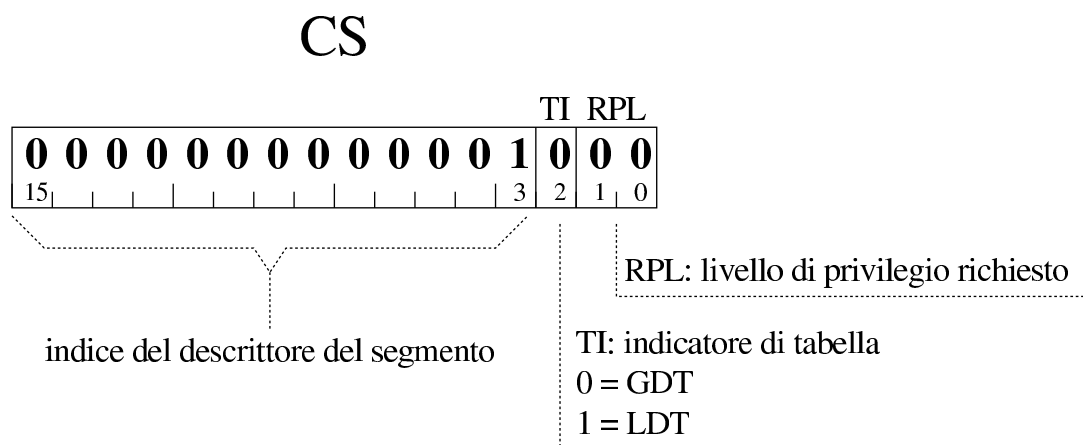
Subito dopo deve essere specificato il valore del registro *CS* (*code segment*) che in questo caso deve corrispondere a un selettore valido per il descrittore del segmento predisposto nella tabella GDT per il codice. In questo caso il valore è 08_{16} , come si può vedere poi dalla figura successiva. Tuttavia, non è possibile assegnare il valore al registro e per ottenere il risultato, si usa un salto incondizionato a lunga distanza (*far jump*) a un simbolo rappresentato da un'etichetta che appare a poca distanza, ma con l'indicazione dell'indirizzo di segmento:

```

...
    jmp $0x08, $flush
flush:
...

```

Figura 83.35. Selettore del segmento codice.



Il listato successivo rappresenta una soluzione completa per l'attivazione della tabella GDT, a partire dall'indirizzo della struttura che ne contiene le coordinate:

```

.globl gdt_load
#
gdt_load:
    enter $0, $0
    .equ gdtr_pointer, 8           # Primo parametro.
    mov  gdtr_pointer(%ebp), %eax  # Copia il puntatore
                                   # in EAX.

    leave
#
    lgdt (%eax)                   # Carica il registro GDTR dalla
#                                 # posizione a cui punta EAX.
#
    mov  $0x10, %eax
    mov  %ax, %ds
    mov  %ax, %es
    mov  %ax, %fs

```

```
    mov    %ax, %gs
    mov    %ax, %ss
    jmp   $0x08, $flush
flush:
    ret
```

Il codice mostrato costituisce una funzione che nel linguaggio C ha il prototipo seguente:

```
gdt_load (void *gdt);
```

Va osservato che l'istruzione '**LEAVE**' viene usata prima di passare all'istruzione '**LGDT**'; diversamente, se si tentasse di mettere dopo l'etichetta del simbolo a cui si salta nel modo descritto (per poter impostare il registro **CS**), l'operazione fallirebbe.

83.5 IDT

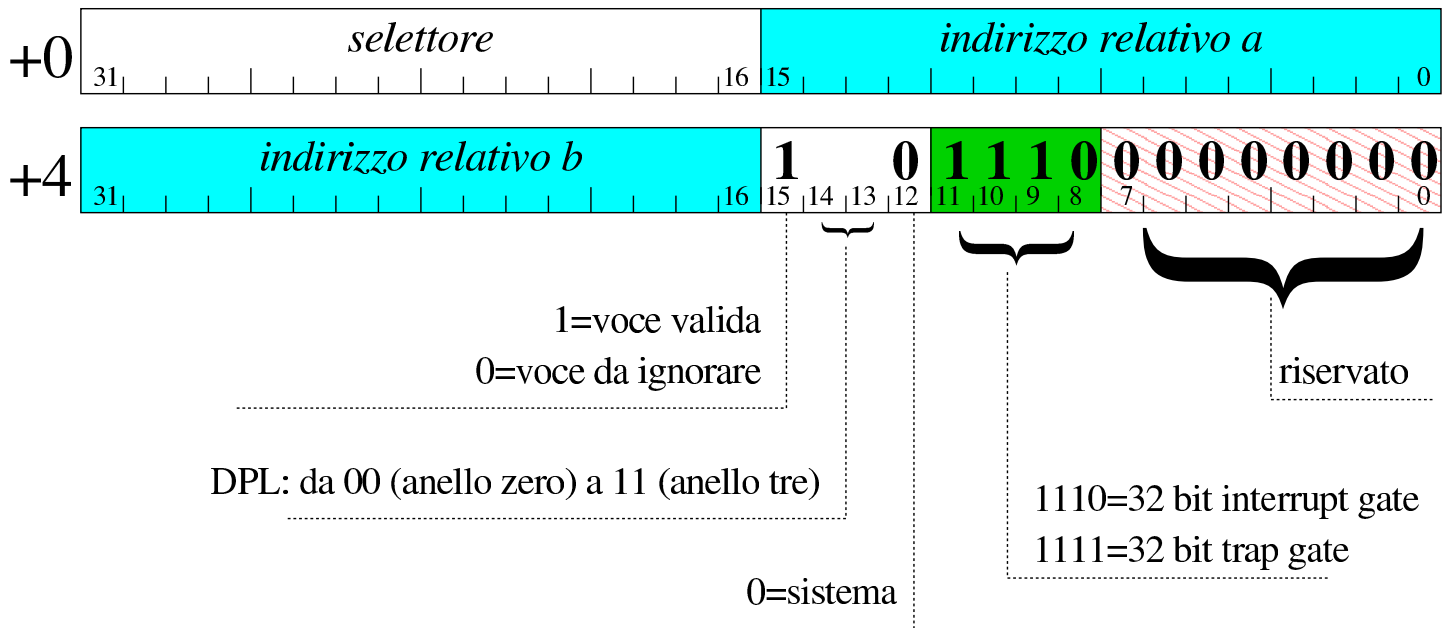
«

La tabella IDT, ovvero *interrupt descriptor table*, serve ai microprocessori x86-32 per conoscere quali procedure avviare al verificarsi delle interruzioni previste. Le interruzioni in questione possono essere dovute a eccezioni (ovvero errori rilevati dal microprocessore stesso), alla chiamata esplicita dell'istruzione che produce un'interruzione software, oppure al verificarsi di interruzioni hardware (IRQ).

Le eccezioni e gli altri tipi di interruzione, vengono associati ognuno a una propria voce nella tabella IDT. Ogni voce della tabella ha un proprio indirizzo di procedura da eseguire al verificarsi dell'interruzione di propria competenza. Tale procedura ha il nome di ISR: *interrupt service routine*.

83.5.1 Struttura della tabella IDT

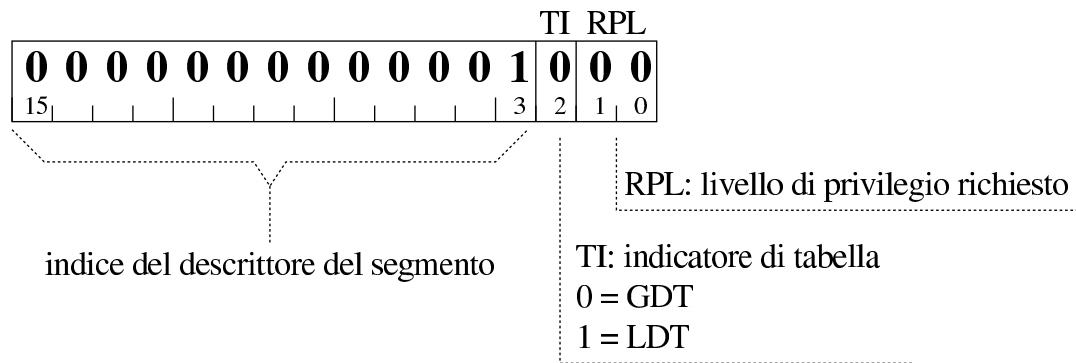
La tabella IDT è costituita da un array di descrittori di interruzione, ognuno dei quali occupa 64 bit. I descrittori possono essere al massimo 256 (da 0 a 255). Nel disegno successivo, viene mostrata la struttura di un descrittore della tabella IDT, prevedendo un accesso a blocchi da 32 bit:



La struttura contiene, in particolare, un selettore di segmento e un indirizzo relativo a tale segmento, riguardante il codice da eseguire quando si manifesta un'interruzione per cui il descrittore è competente (la procedura ISR). L'indirizzo relativo in questione è suddiviso in due parti, da ricomporre in modo abbastanza intuitivo: si prendono le due porzioni dei due blocchi a 32 bit e si uniscono senza dover fare scorrimenti.

Il selettore che si trova nei descrittori della tabella IDT ha la stessa struttura dei selettori usati direttamente con i registri per l'accesso al codice e ai dati. Per i fini degli esempi che vengono mostrati, il livello di privilegi richiesto è zero e la tabella dei descrittori di segmento a cui ci si riferisce è la GDT:

Figura 83.38. Selettore del segmento codice della procedura ISR.



In base a quanto si vede nel disegno e per gli esempi che si fanno nel capitolo, il selettore del segmento codice per le procedure ISR corrisponde a 0008_{16} . Inoltre, negli esempi si fa riferimento esclusivamente a descrittori di tipo *interrupt gate* (a 32 bit).

83.5.2 Codice per la costruzione di una tabella IDT

«

Per costruire una tabella IDT potrebbe essere usata una struttura abbastanza ordinata; tuttavia, il tipo di descrittore e gli altri attributi non potrebbero essere suddivisi come richiederebbe il caso, pertanto qui si preferisce una struttura che si limita a riprodurre due blocchi a 32 bit, come già fatto nella sezione [83.4](#) a proposito della tabella GDT.

```

typedef struct {
    uint32_t  w0;
    uint32_t  w1;
} idt_descriptor_t;
  
```

La funzione successiva riceve come argomento un array di descrittori di una tabella IDT, con l'indicazione dell'indice a cui si vuole fare riferimento e degli attributi che gli si vogliono associare:

```

#include <stdint.h>
static void
idt_descriptor_set (idt_descriptor_t *idt,
  
```



```
        int          descr,
        uint32_t     offset,
        uint32_t     selector,
        uint32_t     type,
        uint32_t     attrib)
{
    //
    // Azzera inizialmente la voce.
    //
    idt[descr].w0 = 0;
    idt[descr].w1 = 0;
    //
    // Indirizzo relativo.
    //
    idt[descr].w0 = idt[descr].w0 | (offset & 0x0000FFFF);
    idt[descr].w1 = idt[descr].w1 | (offset & 0xFFFF0000);
    //
    // Selettore di segmento.
    //
    idt[descr].w0
        = idt[descr].w0 | ((selector << 16) & 0xFFFF0000);
    //
    // Tipo (gate type).
    //
    idt[descr].w1
        = idt[descr].w1 | ((type << 8) & 0x00000F00);
    //
    // Altri attributi.
    //
    idt[descr].w1
        = idt[descr].w1 | ((type << 12) & 0x0000F000);
}
```

Per poter usare questa funzione occorre dichiarare prima l'array che

rappresenta la tabella IDT. Di norma viene creata con tutti 256 descrittori possibili, assicurandosi che inizialmente siano azzerati effettivamente, anche se sarebbe sufficiente azzerare il bit di validità (il bit 15 del secondo blocco a 32 bit):

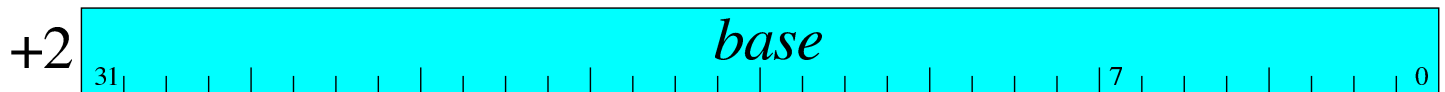
```
...
    static idt_descriptor_t idt[256];
...
    for (descr = 0; descr < 256; descr++)
        {
            idt_descriptor_set (idt, descr, 0, 0, 0, 0);
        }
...
```

Nell'esempio, l'array *idt[]* viene creato specificando l'uso di memoria «statica», nell'ipotesi che ciò avvenga dentro una funzione; diversamente, può trattarsi di una variabile globale senza vincoli particolari.

83.5.3 Attivazione della tabella IDT

«

La tabella IDT può essere collocata in memoria dove si vuole, ma perché il microprocessore la prenda in considerazione, occorre utilizzare un'istruzione specifica con la quale si carica il registro *IDTR* (*IDT register*) a 48 bit. Questo registro non è visibile e si carica con l'istruzione '**LIDT**', la quale richiede l'indicazione dell'indirizzo di memoria dove si articola una struttura contenente le informazioni necessarie. Si tratta precisamente di quanto si vede nel disegno successivo:



In pratica, vengono usati i primi 16 bit per specificare la grandezza complessiva della tabella IDT e altri 32 bit per indicare l'indirizzo in cui inizia la tabella stessa. Tale indirizzo, sommato al valore specificato nel primo campo, deve dare l'indirizzo dell'ultimo byte della tabella stessa.

Dal momento che la dimensione di un descrittore della tabella IDT è di 8 byte, il valore del limite corrisponde sempre a $8 \times n - 1$, dove n è la quantità di descrittori della tabella. Così facendo, si può osservare che gli ultimi tre bit del limite sono sempre impostati a uno.

Nel disegno è stato mostrato chiaramente che il primo campo da 16 bit va considerato in modo separato. Infatti, si intende che l'accesso in lettura o in scrittura vada fatto lì esattamente a 16 bit, perché diversamente i dati risulterebbero organizzati in un altro modo. Pertanto, nel disegno viene chiarito che il campo contenente l'indirizzo della tabella, inizia esattamente dopo due byte. In questo caso, con l'aiuto del linguaggio C è facile dichiarare una struttura che riproduce esattamente ciò che serve per identificare una tabella IDT:

```
#include <stdint.h>
typedef struct {
    uint16_t  limit;
    uint32_t  base;
} __attribute__((packed)) idtr_t;
```

L'esempio mostrato si riferisce all'uso del compilatore GNU C, con il quale è necessario specificare l'attributo *packet*, per fare in modo che i vari componenti risultino abbinati senza spazi ulteriori di allineamento.

Avendo definito la struttura, si può creare una variabile che la utilizza, tenendo conto che è sufficiente rimanga in essere solo fino a quando viene acquisita la tabella IDT relativa dal microprocessore:

```
...  
    idtr_t idtr;  
...
```

Per calcolare il valore che rappresenta la dimensione della tabella, occorre moltiplicare la dimensione di ogni voce (8 byte) per la quantità di voci, sottraendo dal risultato una unità. L'esempio presuppone che si tratti di 256 voci:

```
...  
    idtr.limit = ((sizeof (idt_descriptor_t)) * 256) - 1;  
...
```

L'indirizzo in cui si trova la tabella IDT, può essere assegnato in modo intuitivo:

```
...  
    idtr.base = (uint32_t) &idt[0];  
...
```

Per rendere operativo il contenuto della tabella IDT, quando questa è stata popolata correttamente, va indicato al microprocessore l'indirizzo della struttura che contiene le coordinate della tabella stessa, attraverso l'istruzione '**LIDT**' (*load IDT*). Negli esempi seguenti si utilizzano istruzioni del linguaggio assembler, secondo la sintas-

si di GNU AS; in quello seguente, in particolare, si suppone che il registro *EAX* contenga l'indirizzo in questione:

```
...
    lidt (%eax)  # EAX contiene l'indirizzo della struttura.
...
```

L'attivazione non richiede altro e non ci sono registri da modificare; pertanto, il listato seguente mostra una funzione che provvede a questo lavoro:

```
.globl idt_load
#
idt_load:
    enter $0, $0
    .equ idtr_pointer, 8          # Primo parametro.
    mov idtr_pointer(%ebp), %eax # Copia il puntatore
                                # in EAX.

    leave
#
    lidt (%eax)  # Utilizza la tabella IDT a cui punta EAX.
#
    ret
```

Il codice mostrato costituisce una funzione che nel linguaggio C ha il prototipo seguente:

```
idt_load (void *idtr);
```

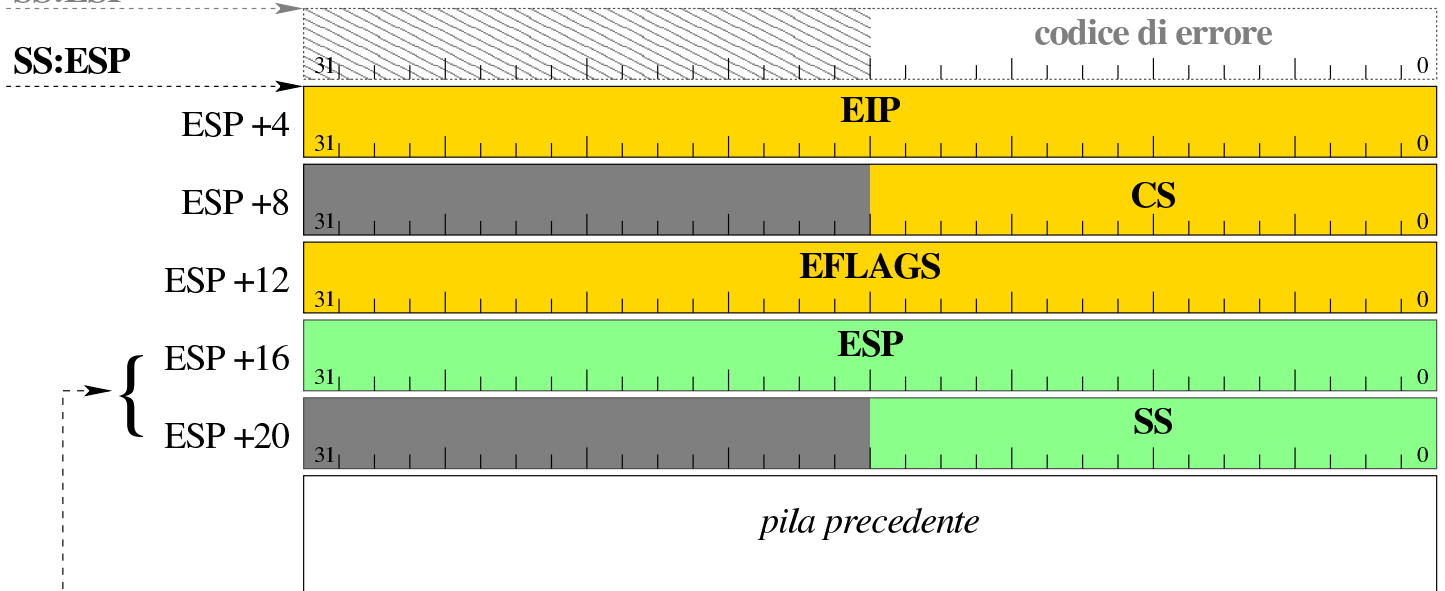
È il caso di ribadire che l'attivazione della tabella IDT va fatta solo dopo che le sue voci sono state compilate con l'indicazione delle procedure di interruzione (ISR) da eseguire.

83.5.4 Lo stato della pila al verificarsi di un'interruzione

«

Al verificarsi di un'interruzione (che coinvolge la consultazione della tabella IDT), il microprocessore accumula alcuni registri sulla pila dell'anello in cui deve essere eseguito il codice delle procedure di interruzione (ISR), come si vede nel disegno successivo, dove la pila viene rappresentata in modo crescente dal basso verso l'alto. Va osservato che i registri *SS* e *ESP* vengono accumulati nella pila solo se i privilegi effettivi cambiano rispetto a quelli del processo da cui si proviene, perché in quel caso, al termine della procedura ISR, occorre ripristinare la pila preesistente; inoltre, quando l'interruzione è causata da un'eccezione prodotta dal microprocessore, in alcuni casi viene accumulato anche un codice di errore.

SS:ESP



questi elementi sono inseriti solo se cambia il livello di privilegio (anello),
 pertanto servono a raggiungere la pila dell'anello di origine

Al termine di una procedura di interruzione, per ripristinare correttamente lo stato dei registri, ovvero per riprendere l'attività sospesa, si usa l'istruzione '**IRET**'.

83.5.5 Bozza di un gestore di interruzioni

Per costruire un gestore di interruzioni è necessario predisporre un po' di codice in linguaggio assembler, dal quale poi è possibile chiamare altro codice scritto con un linguaggio più evoluto. Per poter gestire tutte le interruzioni in modo uniforme, occorre distinguere i casi in cui viene inserito automaticamente un codice di errore nella pila dei dati, da quelli in cui ciò non avviene; pertanto, nell'esempio viene inserito un codice nullo di errore quando non si prevede tale inserimento a cura del microprocessore, in modo da avere la stessa struttura della pila dei dati. Lo schema usato in questo listato è sostanzialmente conforme a un esempio analogo che appare nel documento *Bran's kernel development tutorial*, di Brandon Friesen, citato alla fine del capitolo.

```
.extern interrupt_handler
#
.globl isr_0
.globl isr_1
...
.globl isr_254
.globl isr_255
#
isr_0:          # division by zero exception
    cli
    push $0     # Codice di errore fittizio.
    push $0     # Numero di procedura ISR.
    jmp isr_common
#
isr_1:          # debug exception
    cli
    push $0     # Codice di errore fittizio.
    push $1     # Numero di procedura ISR.
```

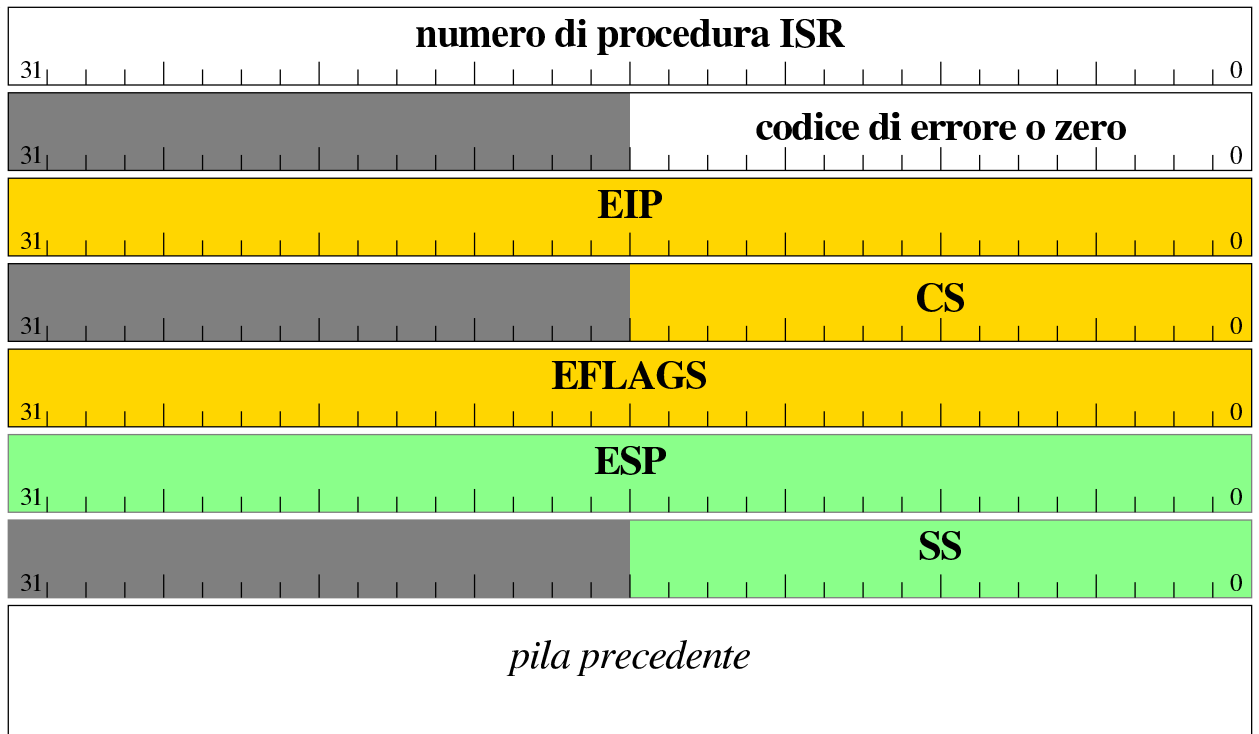
```
    jmp isr_common
...
isr_8:          # double fault exception
    cli
    #
    push $8     # Numero di procedura ISR.
    jmp isr_common
...
#
isr_32:        # IRQ 0: timer
    cli
    push $0     # Codice di errore fittizio.
    push $32   # Numero di procedura ISR.
    jmp isr_common
#
isr_33:        # IRQ 1: tastiera
    cli
    push $0     # Codice di errore fittizio.
    push $1    # Numero di procedura ISR.
    jmp isr_common
...
isr_47:        # IRQ 15: canale IDE secondario
    cli
    push $0     # Codice di errore fittizio.
    push $15   # Numero di procedura ISR.
    jmp isr_common
...
#
isr_common:
    pushl %gs
    pushl %fs
    pushl %es
    pushl %ds
    pushl %edi
```



```
    pushl %esi
    pushl %ebp
    pushl %ebx
    pushl %edx
    pushl %ecx
    pushl %eax
    #
    call interrupt_handler
    #
    popl %eax
    popl %ecx
    popl %edx
    popl %ebx
    popl %ebp
    popl %esi
    popl %edi
    popl %ds
    popl %es
    popl %fs
    popl %gs
    add $4, %esp      # Espelle il numero di procedura ISR.
    add $4, %esp      # Espelle il codice di errore (reale o
                     # fittizio).
    #
    iret              # ripristina EIP, CS, EFLAGS, SS
                     # e conclude la procedura.
```

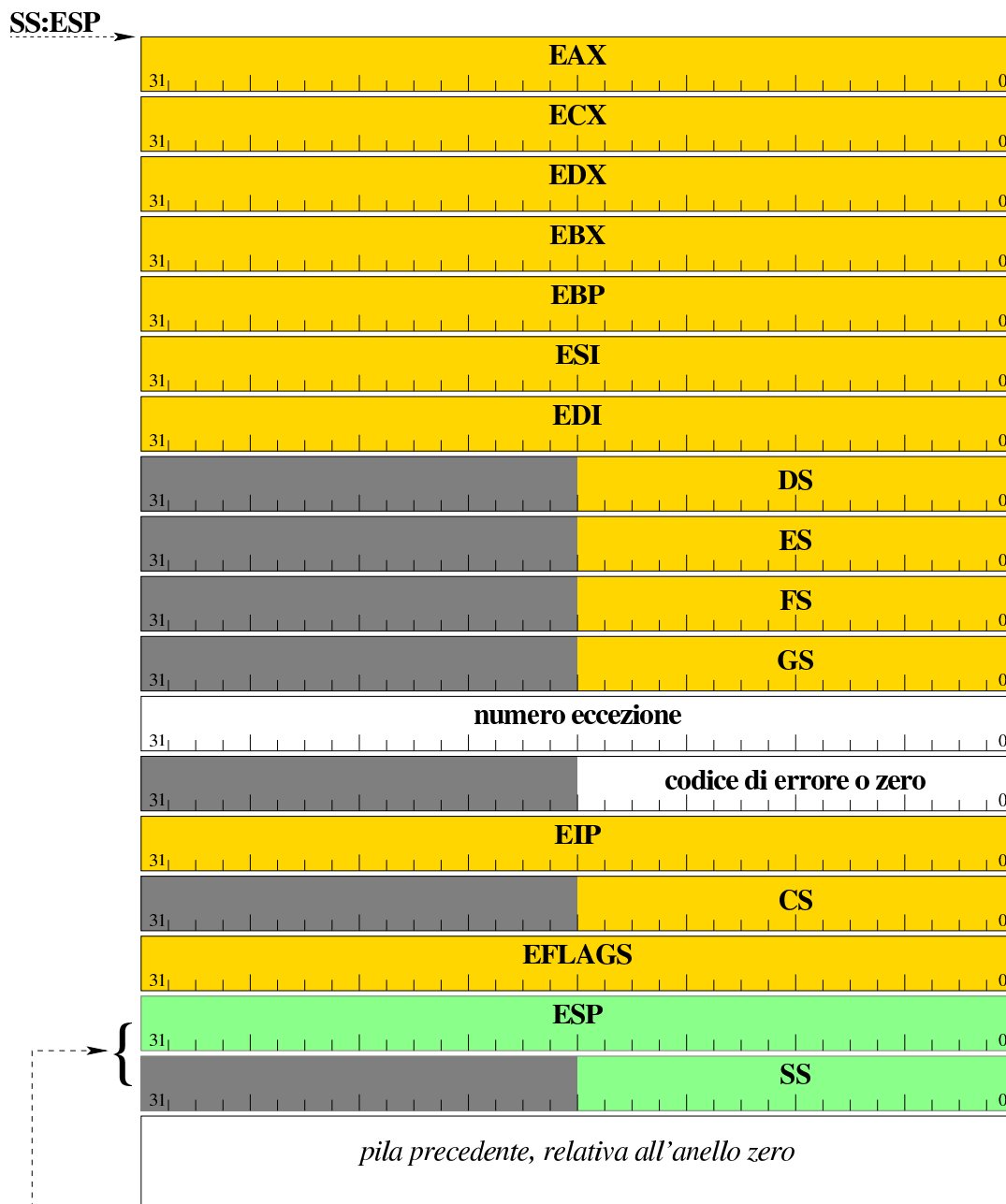
Come si può vedere, quando viene chiamata una procedura che non prevede l'esistenza di un codice di errore, come nel caso di *isr_0()*, al suo posto viene aggiunto un valore fittizio, mentre quando il codice di errore è previsto, come nel caso di *isr_8()*, questo inserimento nella pila viene a mancare. Prima di eseguire il codice che inizia a partire da *isr_common()*, lo stato della pila è il seguente:

SS:ESP →



questi elementi sono inseriti solo se cambia il livello di privilegio (anello),
 pertanto servono a raggiungere la pila dell'anello di origine

Il codice che si trova a partire da *isr_common()* serve a preparare la chiamata di una funzione, scritta presumibilmente in C, pertanto si procede a salvare i registri; qui si includono anche quelli di segmento, per maggiore scrupolo. Al momento della chiamata, la pila ha la struttura seguente:



questi elementi sono inseriti solo se cambia il livello di privilegio (anello),
 pertanto servono a raggiungere la pila dell'anello di origine

In base a questo contenuto della pila, una funzione scritta in C per il trattamento dell'eccezione, può avere il prototipo seguente:

```
void interrupt_handler (uint32_t eax,
                        uint32_t ecx,
                        uint32_t edx,
                        uint32_t ebx,
                        uint32_t ebp,
                        uint32_t esi,
                        uint32_t edi,
                        uint32_t ds,
                        uint32_t es,
                        uint32_t fs,
                        uint32_t gs,
                        uint32_t isr,
                        uint32_t error,
                        uint32_t eip,
                        uint32_t cs,
                        uint32_t eflags, ...);
```

I puntini di sospensione riguardano la possibilità, eventuale, di accedere anche ai valori di *ESP* e *SS*, quando il contesto prevede il loro accumulo.

Una volta definita in qualche modo la funzione esterna che tratta le interruzioni, le procedure ISR del file che le raccoglie (quello mostrato in linguaggio assembleatore) servono ad aggiornare la tabella IDT, la quale inizialmente è stata azzerata in modo da annullare l'effetto dei suoi descrittori. Nel listato seguente, *idt* è l'array di descrittori che forma la tabella IDT:

```

idt_descriptor_set (idt, 0, (uint32_t) isr_0, 0x08, 0xE, 0x8);
idt_descriptor_set (idt, 1, (uint32_t) isr_1, 0x08, 0xE, 0x8);
idt_descriptor_set (idt, 2, (uint32_t) isr_2, 0x08, 0xE, 0x8);
...
...

```

Le procedure ISR inserite nella tabella IDT devono essere solo quelle che sono operative effettivamente; per le altre è meglio lasciare i valori a zero.

83.5.6 Una funzione banale per il controllo delle interruzioni

Viene mostrato un esempio banale per la realizzazione della funzione *interrupt_handler()*, a cui si fa riferimento nella sezione precedente. Si parte dal presupposto di poter utilizzare la funzione *printf()*.

```

#include <stdint.h>
#include <inttypes.h>
#include <stdio.h>
void
interrupt_handler (uint32_t eax, uint32_t ecx, uint32_t edx,
                  uint32_t ebx, uint32_t ebp, uint32_t esi,
                  uint32_t edi, uint32_t ds,  uint32_t es,
                  uint32_t fs,  uint32_t gs,  uint32_t isr,
                  uint32_t error, uint32_t eip,
                  uint32_t cs, uint32_t eflags, ...)
{
    printf ("ISR %3" PRIi32 " ", error %08" PRIX32 "\n", isr,
           error);
}

```

83.5.7 Privilegi e protezioni

«

Negli esempi mostrati, ogni riferimento a privilegi di esecuzione e di accesso si riferisce sempre all'anello zero, pertanto non si possono creare problemi. Ma la realtà si può presentare in modo più complesso e va osservato che il livello corrente dei privilegi (CPL), nel momento in cui si verifica un'interruzione, non è prevedibile.

La prima cosa da considerare è il livello di privilegio del descrittore (DPL) del segmento codice in cui si trova la procedura ISR, il quale deve essere numericamente inferiore o uguale al livello corrente (CPL) precedente all'interruzione. Di conseguenza, è normale attendersi che le interruzioni comuni siano gestite da procedure ISR collocate in codice con un livello di privilegio del descrittore di segmento pari a zero.

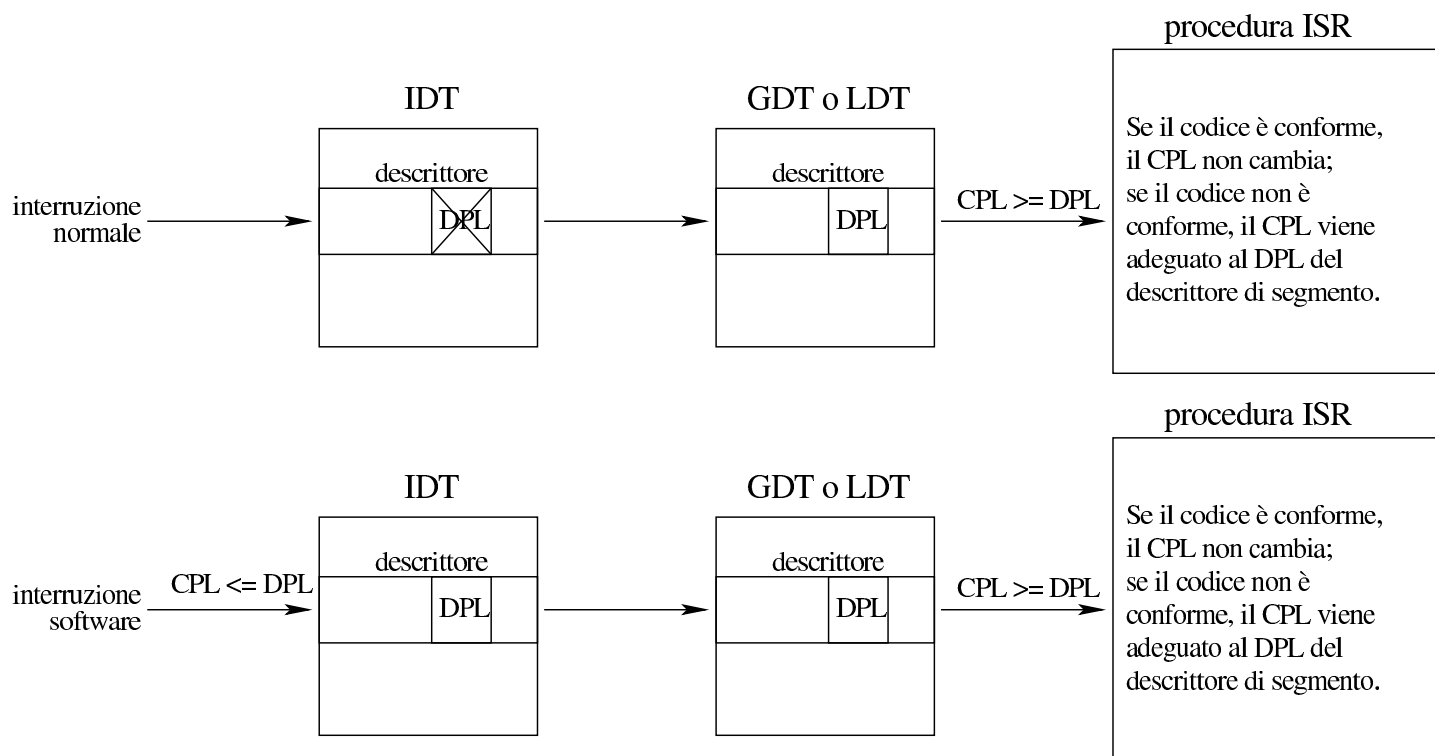
Nel selettore del descrittore di interruzione non viene considerato il valore RPL, anche se è bene che questo sia azzerato.

Il livello di privilegio del descrittore (DPL) di interruzione viene **considerato solo in presenza di un'interruzione prodotta da software**, ovvero per un'interruzione prodotta volontariamente con le istruzioni apposite. In tal caso, il livello di privilegio corrente (CPL) del processo che la genera deve essere numericamente inferiore o uguale a quello del descrittore di interruzione. Pertanto, mettendo un valore DPL per il descrittore di interruzione pari a zero, si impedisce ai processi non privilegiati di far scattare le interruzioni in modo volontario.

Se il segmento codice dove si trova la procedura ISR è di tipo «non conforme», se il livello di privilegio corrente precedente è diverso (in questo contesto può essere solo numericamente maggiore), allora

viene modificato e adeguato a quello del segmento codice raggiunto, con l'aggiunta dello scambio della pila di dati. Se invece il segmento codice dove si trova la procedura ISR è di tipo «conforme», non può avvenire alcun miglioramento di privilegi. Tra le altre cose, questa scelta ha anche delle ripercussioni per ciò che riguarda l'accesso ai dati: **il gestore di interruzione che abbia la necessità di accedere a dati che siano al di fuori della pila, deve trovarsi a funzionare all'interno di un segmento codice «non conforme», con privilegi DPL pari a zero**; diversamente (se si accontenta della pila, ovvero di variabili automatiche proprie), può funzionare semplicemente in un segmento codice conforme.

Figura 83.55. Verifica dei privilegi per l'esecuzione di una procedura ISR, a partire da un'interruzione.



83.6 Gestione delle interruzioni

«

Le interruzioni possono essere fondamentalmente di tre tipi: eccezioni prodotte dal microprocessore, interruzioni hardware (IRQ) e interruzioni prodotte attraverso istruzioni (ovvero interruzioni software). Le interruzioni vanno associate ai descrittori della tabella IDT (*interrupt descriptor table* in modo appropriato).

83.6.1 Eccezioni

«

Le **eccezioni** sono eventi che si manifestano in presenza di errori, di cui è competente direttamente il microprocessore. Le eccezioni sono numerate e sono già associate alla tabella IDT con gli stessi numeri: l'eccezione ***n*** è abbinata al descrittore ***n*** della tabella. Sono previste 32 eccezioni, numerate da 0 a 31, pertanto i descrittori da 0 a 31 della tabella IDT sono già impegnati per questa gestione e vanno utilizzati coerentemente in tale direzione.

Va ricordato che in presenza di alcuni tipi di eccezione, il microprocessore accumula nella pila un codice di errore, pertanto, per uniformare le procedure ISR (*interrupt service routine*), occorre tenere conto dei casi in cui tale informazione è già inserita nella pila, rispetto a quelli dove questa non c'è ed è bene aggiungere un valore fittizio per coerenza.

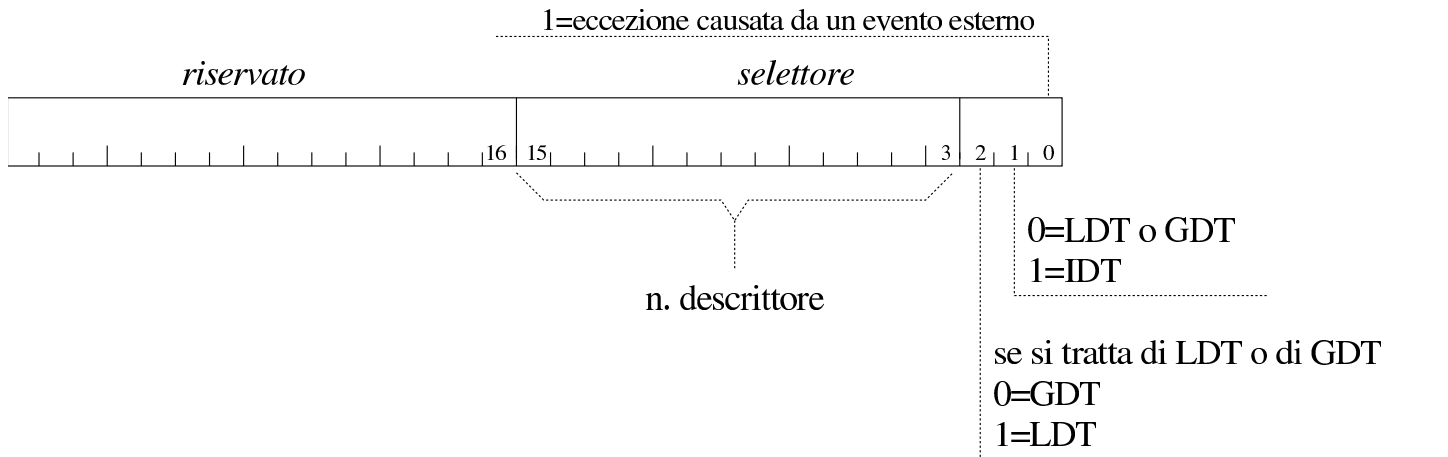
Tabella 83.56. Elenco delle eccezioni.

Eccezione	Codice di errore aggiunto sulla pila?	Definizione dell'eccezione
0	no	<i>division by zero</i>
1	no	<i>debug</i>

Eccezione	Codice di errore aggiunto sulla pila?	Definizione dell'eccezione
2	no	<i>non maskable interrupt</i>
3	no	<i>breakpoint</i>
4	no	<i>into detected overflow</i>
5	no	<i>out of bounds</i>
6	no	<i>invalid opcode</i>
7	no	<i>no coprocessor</i>
8	SÌ	<i>double fault</i>
9	no	<i>coprocessor segment overrun</i>
10	SÌ	<i>bad TSS</i>
11	SÌ	<i>segment not present</i>
12	SÌ	<i>stack fault</i>
13	SÌ	<i>general protection fault</i>
14	SÌ	<i>page fault</i>
15	no	<i>unknown interrupt</i>
16	no	<i>coprocessor fault</i>
17	no	<i>alignment check exception</i>
18	no	<i>machine check exception</i>
da 19 a 31	no	eccezioni riservate per il futuro

Il codice di errore che inserisce il microprocessore sulla pila, quando si verificano le eccezioni che lo prevedono, ha una struttura variabile, in base al tipo di eccezione. Lo schema della figura successiva è abbastanza comune e riguarda un errore per il quale viene fatto riferimento a un selettore (per la tabella GDT, LDT o IDT, in base al contesto).

Figura 83.57. Codice di errore prodotto da alcune eccezioni.



Come si può intendere dal disegno, a seconda dei valori dei bit 1 e 2, il selettore va inteso riguardare una voce della tabella GDT, oppure di una tabella LDT o della tabella IDT stessa.

Quando il codice di errore è completamente a zero, almeno nei primi 16 bit meno significativi, vuol dire che non riguarda un problema collegabile a una voce di una delle tabelle IDT, LDT o GDT.

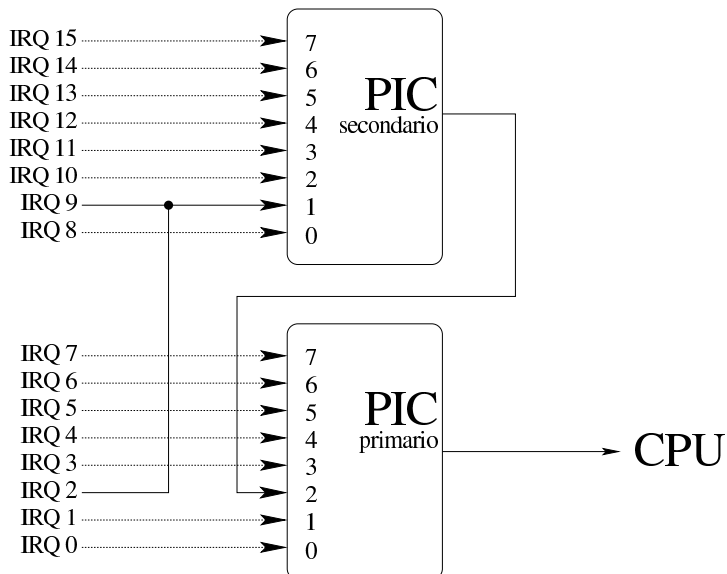
83.6.2 PIC e rimappatura delle interruzioni

«

Per affrontare al gestione delle interruzioni hardware, occorre prima premettere una breve introduzione, a causa del fatto che non si tratta di una funzione gestita autonomamente dal microprocessore.

Secondo la tradizione dell'architettura IBM PC/AT, per raccogliere le interruzioni hardware dell'elaboratore sono utilizzati due integrati, chiamati generalmente PIC, ovvero *programmable interrupt controller*, collegati assieme in modo da poter recepire complessivamente quindici interruzioni hardware differenti. Per la precisione, il PIC secondario, se riceve un'interruzione, va a provocare un IRQ 2 nel PIC primario; pertanto, se si ricevono interruzioni tra IRQ 8 e IRQ 15, si ottiene anche un'interruzione su IRQ 2. Dal momento

che IRQ 2 è impegnato, quello che sarebbe il segnale di IRQ 2 viene ridiretto a IRQ 9. Il disegno seguente serve solo a chiarire il concetto, dal momento che i collegamenti effettivi sono più complessi:



Le interruzioni hardware, o «IRQ», vanno abbinate a interruzioni della tabella IDT, per poterle gestire in qualche modo. Purtroppo, originariamente esiste già un abbinamento, ma incompatibile con quello delle eccezioni del microprocessore; pertanto, va rifatta la mappa di trasformazione.

Per comunicare con i due PIC e per riprogrammarli, esistono delle porte di comunicazione: 20_{16} e 21_{16} per il PIC principale; $A0_{16}$ e $A1_{16}$ per il PIC secondario. La procedura per rimappare i PIC richiede la scrittura di diversi valori che, a seconda dei casi, prendono il nome di «ICW» (*initialization command word*) e «OCW» (*operation command word*). La funzione seguente, scritta in linguaggio C, permette la rimappatura dei due PIC e abilita automaticamente tutte le interruzioni hardware (che altrimenti potrebbero anche essere mascherate).

```
#include <stdio.h>
void
```

```
irq_remap (unsigned int offset_1, unsigned int offset_2)
{
    //
    // PIC_P è il PIC primario o «master»;
    // PIC_S è il PIC secondario o «slave».
    //
    // Quando si manifesta un IRQ che riguarda il PIC
    // secondario, il PIC primario riceve IRQ 2.
    //
    // ICW = initialization command word.
    // OCW = operation command word.
    //
    printf ("kernel: PIC "
           "(programmable interrupt controller) remap: ");

    outb (0x20, 0x10 + 0x01); // Inizializzazione: 0x10
    outb (0xA0, 0x10 + 0x01); // significa che si tratta di
    printf ("ICW1");          // ICW1; 0x01 significa che si
                              // deve arrivare fino a ICW4.

    outb (0x21, offset_1);    // ICW2: PIC_P a partire da
    outb (0xA1, offset_2);    // «offset_1»; PIC_S a partire
    printf ("", ICW2);        // da «offset_2».

    outb (0x21, 0x04);        // ICW3 PIC_P: IRQ2 pilotato da PIC_S.
    outb (0xA1, 0x02);        // ICW3 PIC_S: pilota IRQ2 di PIC_P.
    printf ("", ICW3);

    outb (0x21, 0x01);        // ICW4: si precisa solo la modalità
    outb (0xA1, 0x01);        // del microprocessore; 0x01 = 8086.
    printf ("", ICW4);

    outb (0x21, 0x00);        // OCW1: azzera la maschera in modo
    outb (0xA1, 0x00);        // da abilitare tutti i numeri IRQ.
    printf ("", OCW1.\n");
```

```
}
```

Nel corso del procedimento di rimappatura delle interruzioni, è necessario fare delle brevissime pause, per dare il tempo ai PIC di recepire le informazioni; a tale proposito sono state aggiunte delle istruzioni che visualizzano il progresso nelle varie fasi di rimappatura. Le sigle che appaiono nei commenti del listato, richiamano i termini usati per identificare i valori che sono attribuiti alle porte, in modo da poter ritrovare nella documentazione dei PIC il significato che hanno.

La funzione proposta nell'esempio riceve due argomenti, corrispondenti allo spostamento delle interruzioni del primo e del secondo PIC. Per esempio, ammesso di voler spostare le interruzioni del primo PIC a partire da 32_{10} e quelle del secondo PIC a partire da 40_{10} , in modo da utilizzare esattamente le voci della tabella IDT successive a quelle delle eccezioni, basta usare la funzione nel modo seguente:

```
...  
    irq_remap (32, 40);  
...
```

83.6.3 Procedura generalizzata per la gestione delle interruzioni

Nella sezione [83.5.5](#) appare il codice iniziale, in linguaggio assembler, per la gestione delle interruzioni. A partire da lì viene richiamata la funzione *interrupt_handler()*, dalla quale è possibile risalire al numero di procedura ISR da attivare. Per rendere intercambiabili le funzioni che gestiscono specificatamente ogni singola interruzione, potrebbe essere conveniente predisporre un array di puntatori



a funzione, ma per comodità viene dichiarato semplicemente come array di puntatori generici, inizialmente azzerati:

```
...
void *isr_func[256] = {0};
...
```

Le funzioni che si associano agli elementi dell'array devono essere tali da poter gestire l'interruzione di propria competenza. Per esempio, *isr_func[0]* deve essere il puntatore di una funzione in grado di gestire l'interruzione derivante dall'eccezione *divide error*.

Ammesso di avere popolato correttamente l'array *isr_func[]*, la funzione *interrupt_handler()* potrebbe essere fatta così:

```
#include <stdint.h>
#include <inttypes.h>
#include <stdio.h>
void
interrupt_handler (uint32_t eax, uint32_t ecx, uint32_t edx,
                  uint32_t ebx, uint32_t ebp, uint32_t esi,
                  uint32_t edi, uint32_t ds,  uint32_t es,
                  uint32_t fs,  uint32_t gs,  uint32_t isr,
                  uint32_t error, uint32_t eip,
                  uint32_t cs, uint32_t eflags, ...)
{
    if (isr > 255)
    {
        printf ("kernel: %s: error: cannot handle "
               "ISR %" PRIi32 "!\n",
               __func__, isr);
        return;
    }

    //
    // La variabile handler è un puntatore a funzione che ha
```

```
// due parametri di tipo «unsigned int» a 32 bit e
// restituisce «void».
//
void (*handler) (uint32_t isr, uint32_t error);
//
// Carica la funzione associata al numero ISR.
//
handler = isr_func[isr];
//
// Se il puntatore a funzione è diverso da NULL, allora
// procede.
//
if (handler)
{
    handler (isr, error);
}
//
// Se si tratta di un'interruzione hardware, occorre
// informare i PIC coinvolti che l'elaborazione è
// terminata, attraverso un messaggio «EOI».
//
if (isr >= 40 && isr <= 47)
{
    // PIC secondario.
    outb (0xA0, 0x20);
}
//
if (isr >= 32 && isr <= 47)
{
    // Il PIC primario è coinvolto sempre.
    outb (0x20, 0x20);
}
}
```

Come si vede, per semplificare il tutto, le funzioni che devono elaborare le interruzioni devono avere un prototipo di questo tipo:

```
#include <stdint.h>
void nome_funzione (uint32_t isr, uint32_t error);
```

Una funzione generica, anche se poco graziosa, per il trattamento delle eccezioni potrebbe essere fatta così:

```
#include <inttypes.h>
#include <stdio.h>
void
exception_handler (uint32_t isr, uint32_t error)
{
    printf ("kernel: exception %" PRIi32 " ", "
           "error %04" PRIx32 "!\n",
           isr, error);

    //
    // Blocca tutto.
    //
    for (;;)
}
```

Per associare la funzione alle prime 32 voci dell'array *isr_func()*, si potrebbe procedere così:

```
...
int i;
...
for (i = 0; i < 256; i++)
{
    isr_func[i] = exception_handler;
}
...
```


Per quanto riguarda le funzioni che devono gestire le interruzioni di origine hardware, bisogna ricordare che il valore del parametro *isr* non dà il numero IRQ, ma se fosse necessario calcolarlo basterebbe sottrarre il numero 32 da quello del numero della voce ISR originale.

83.6.4 Attivazione

In precedenza è stato mostrato come si attiva la tabella IDT, attraverso l'istruzione '**LIDT**', ma è evidente che questo va fatto solo dopo che la tabella IDT è stata predisposta e che sono state preparate le funzioni per la gestione delle interruzioni (quelle che si vogliono gestire). Ciò che rimane, ammesso di essere pronti a gestire le interruzioni hardware, è l'attivazione di queste interruzioni, con l'istruzione '**STI**' del linguaggio assembler.

83.7 Gestione del temporizzatore: PIT, ovvero «programmable interval timer»

Negli elaboratori con architettura IBM PC/AT, è previsto un temporizzatore costituito originariamente da un integrato programmabile, contenente tre contatori: uno associato a IRQ 0, uno associato a qualche funzione particolare, dipendente dall'organizzazione dell'hardware, un altro associato all'altoparlante interno. Questo integrato è noto con la sigla PIT, ovvero *programmable interval timer*.

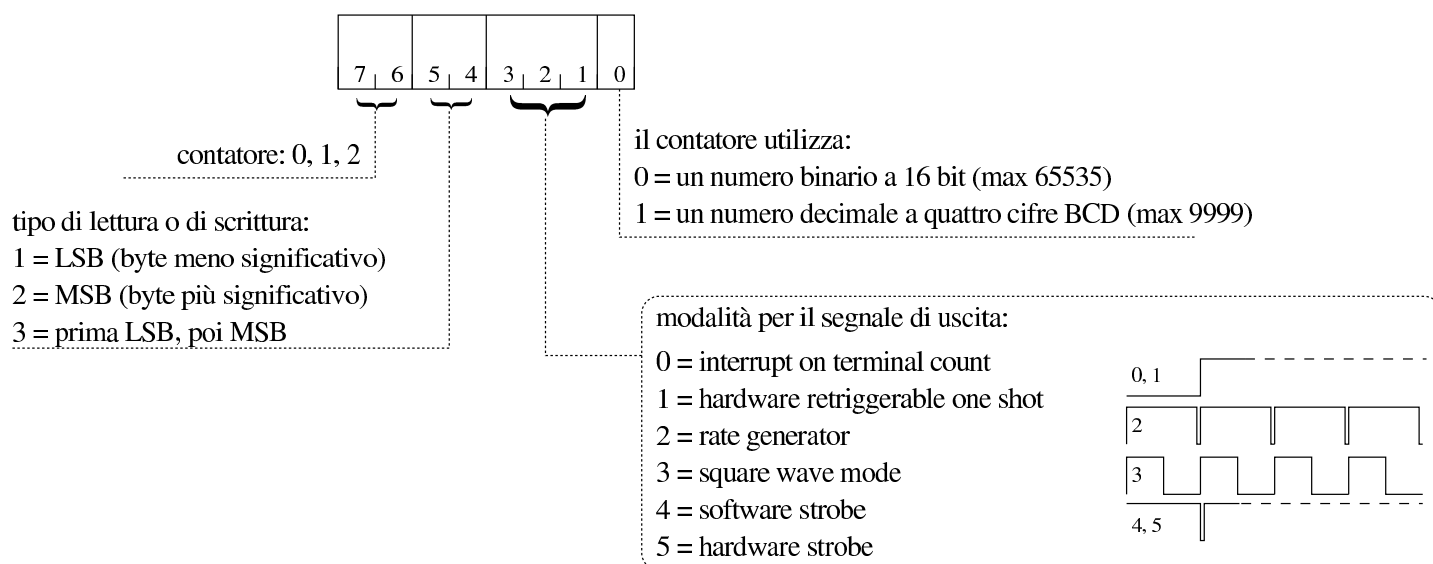
Questo integrato, o comunque ciò che ne fa la funzione, conta degli impulsi provenienti a una frequenza stabilita e, a seconda di come viene programmato, produce un risultato differente nelle sue tre uscite. Per esempio può generare un'onda quadra a una frazione della frequenza ricevuta in ingresso, oppure può emettere altri tipi di

segnali, sempre tenendo in considerazione il risultato del conteggio degli impulsi in ingresso.

Per quanto riguarda la gestione del temporizzatore, ovvero della frequenza con cui si vuole ottenere un'interruzione IRQ 0, generalmente si programma il PIT per produrre un'onda quadra.

Secondo lo standard dell'architettura IBM PC/AT, la frequenza che produce gli impulsi in ingresso del PIT è a 1,19 MHz circa. Più precisamente si tratta di $3\,579\,545/3$ Hz.

La programmazione del PIT avviene inviando un comando (*command word*, o CW), costituito da un byte, alla porta 43_{16} , con il quale, in particolare, si specifica il contatore a cui ci si vuole riferire. Successivamente, a seconda del comando inviato, possono essere trasmessi altri valori alla porta riservata specificatamente per il contatore a cui si è interessati. Il contatore zero che serve a produrre le interruzioni IRQ 0, riceve questi valori dalla porta 40_{16} , mentre la porta 42_{16} è quella del contatore tre, associato all'altoparlante interno (il contatore uno sarebbe associato alla porta 41_{16} , ma in pratica non può essere utilizzato).

Figura 83.65. Comando da inviare alla porta 43₁₆.

La figura appena apparsa schematizza in che modo va composto o interpretato il comando da inviare al PIT. Per quanto riguarda la modalità di funzionamento, quella che serve per generare le interruzioni è la numero 3 (onda quadra); per conoscere il significato delle altre modalità si possono consultare i documenti citati alla fine del capitolo. Il resto delle componenti di un comando dovrebbe essere abbastanza comprensibile, ma vale la pena di riassumere brevemente. I primi due bit più significativi indicano il contatore a cui si vuole fare riferimento. Altri due bit indicano cosa deve essere trasmesso, successivamente al comando, attraverso la porta dei dati: un solo byte, a scelta tra il meno significativo o il più significativo, oppure entrambi i byte, a cominciare da quello meno significativo. Altri tre bit definiscono la modalità. Per quanto riguarda il senso del bit meno significativo, occorre considerare che il contatore degli impulsi ricevuti in ingresso può utilizzare un valore a 16 bit (cosa che si fa normalmente), oppure un numero a sole quattro cifre in base dieci (i 16 bit del contatore verrebbero divisi in quattro gruppi da quattro bit, ognuno dei quali viene usato esclusivamente per rappresentare

valori da zero a nove).

Per programmare il contatore zero, in modo che generi una certa frequenza (purché inferiore a 1,19 MHz), si usa normalmente il comando 36_{16} , il quale: seleziona il contatore zero; stabilisce che il valore da comunicare successivamente viene trasmesso usando due byte (prima quello meno significativo, poi quello più significativo); richiede una modalità di funzionamento a onda quadra; richiede di utilizzare il contatore in modo binario, a 16 bit. Successivamente al comando si usa il valore che rappresenta il divisore della frequenza di 1,19 MHz. Per esempio, volendo generare una frequenza vicina a 100 Hz, dopo aver inviato il comando 36_{16} alla porta 43_{16} , occorre inviare il valore 11931_{10} , separandolo in due byte, alla porta 40_{16} .

Va osservato che il valore del divisore può utilizzare al massimo 16 bit complessivamente, partendo da uno (lo zero non è ammissibile per ovvi motivi). Pertanto, si può dividere la frequenza di ingresso al massimo di 65535 volte.

Segue l'esempio di una funzione con la quale si programma la frequenza delle interruzioni IRQ 0, ma senza verificare che il valore richiesto sia valido:

```
void
timer_freq (int freq)
{
    int input_freq = 1193181;
    int divisor = input_freq / freq;
    outb (0x43, 0x36);           // CW: «command word».
    outb (0x40, divisor & 0x0F); // LSB: byte inferiore del
                                // divisore.
    outb (0x40, divisor / 0x10); // MSB: byte superiore del
                                // divisore.
}
```

Se il PIT non viene riprogrammato, inizialmente lo si trova configurato in modo da generare una frequenza (a onda quadra) di 18,222 Hz che è quella più bassa possibile.

83.8 Tastiera PS/2

La tastiera PS/2 di un elaboratore IBM PC/AT produce un'interruzione ogni volta che si preme o si rilascia un tasto, quindi si può leggere tale codice dalla porta 60_{16} . Per la precisione, dalla porta 60_{16} si può leggere un solo byte alla volta, mentre ci sono situazioni in cui i codici generati dalla pressione o dal rilascio dei tasti sono formati da una sequenza di più byte; pertanto, la tastiera possiede una propria memoria tampone, dalla quale si può leggere sequenzialmente.

Il funzionamento della tastiera può essere configurato, inviando, a porte differenti, dei comandi che qui non vengono trattati; tuttavia la documentazione annotata nella bibliografia riporta tali informazioni.

Il codice che si può leggere attraverso la porta 60_{16} è definito *scancode*, ma ne esistono normalmente tre versioni, di cui quella standard (predefinita) è la seconda. Per conoscere i codici generati dalla ta-

stiera si può utilizzare il programma **'showkey'**, con l'opzione **'-s'**, da un sistema GNU/Linux. Con l'aiuto di questo programma si può anche comprendere bene come vengano generati i codici e l'effetto della ripetizione automatica.

Come regola generale va osservato che i tasti premuti producono un codice inferiore o uguale a 127_{10} (ovvero $7F_{16}$, oppure 1111111_2), mentre i tasti rilasciati producono il valore corrispondente alla somma del codice di pressione più 128_{10} (ovvero 80_{16} , oppure 1000000_2). In pratica, si riconosce il rilascio di un tasto per il fatto che il bit più significativo è impostato a uno.

Le sequenze multiple di alcuni tasti servono normalmente a distinguerli rispetto ad altri equivalenti, inserendo normalmente il codice $E0_{16}$. Per esempio, il tasto [*Ctrl*] sinistro produce il codice $1D_{16}$ alla pressione e $9D_{16}$ al rilascio, mentre il tasto [*Ctrl*] destro produce $E0_{16} 1D_{16}$ alla pressione e $E0_{16} 9D_{16}$ al rilascio. Pertanto se si vuole semplificare l'interpretazione dei tasti premuti dalla tastiera, si potrebbero ignorare i codici speciali che servono per le sequenze multiple.

Figura 83.67. Mappa dei codici della tastiera. Sulla parte superiore sinistra appare la sequenza generata dalla pressione del tasto, mentre sulla parte inferiore destra appare quella associata al rilascio del tasto. Le sequenze sono espresse in esadecimale.

01 81	3B BB	3C BC	3D BD	3E BE	3F BF	40 C0	41 C1	42 C2	43 C3	44 C4	57 D7	58 D8	E0,2A,E0,37 E0,AA,E0,B7	46 C6	E1,1D,A5 E1,9D,C5							
29 A9	02 1	03 2	04 3	05 4	06 5	07 6	08 7	09 8	0A 9	0B A	0C B	0D C	0E D	E0,52 E0,D2	E0,47 E0,C7	E0,49 E0,C9	45 C5	E0,35 E0,B5	37 B7	4A CA		
0F tab	10 8F	11 q	12 90	13 w	14 91	15 e	16 92	17 r	18 93	19 t	1A 94	1B y	1C 95	E0,53 E0,D3	E0,4F E0,CF	E0,51 E0,D1	47 C7	48 C8	49 C9	4E CA		
3A ↑	1E BA	1F a	20 9E	21 s	22 9F	23 d	24 A0	25 f	26 A1	27 g	28 A2	29 h	2A A3	2B A4	2C A5	2D A6	2E A7	2F A8	30 AB	31 ←	32 9C	
2A ↑	AA	38 B8	2C z	2D AC	2E x	2F AD	30 c	31 AE	32 AF	33 v	34 B0	35 b	36 B1	E0,48 E0,C8	E0,38 E0,B8	E0,5C E0,DC	E0,5D E0,DD	E0,1D E0,9D	4B CB	4C CC	4D CD	4E CE
1D *	9D	E0,5B E0,DB	38 B8	39 ↵	B9	E0,38 E0,B8	E0,5C E0,DC	E0,5D E0,DD	E0,1D E0,9D	E0,4B E0,CB	E0,50 E0,D0	E0,4D E0,CD	4F CF	50 D0	51 D1	E0,1C E0,9C	52 D2	53 D3	↵			

83.9 Gestione di dischi PATA

Qui si descrive come accedere a unità a disco ATA (*AT attachment*) con cavo parallelo (PATA), da un elaboratore con architettura conforme al IBM PC, secondo la modalità PIO (*Programmed input-output*), con la quale si impegna direttamente la CPU per il trasferimento dei dati. La modalità di trasferimento PIO è logicamente quella più costosa per il sistema, ma è anche la più semplice da ottenere in termini di programmazione. Non si considerano unità ATAPI (*AT attachment packet interface*) e comunque ci si sofferma prevalentemente sulla modalità di accesso LBA28. (si veda eventualmente anche la sezione 9.3 sulle unità PATA.)

Le unità a disco PATA, si connettono attraverso un cavo piatto (piattina) a un bus; su tale cavo si possono collegare due dischi: uno dei due viene chiamato *master* e l'altro *slave*. In pratica, la distinzione delle unità ATA attraverso queste denominazioni non è appropriata, perché non esistono ruoli sostanzialmente differenti; piuttosto si

tratta solo di distinguere le due unità. È il caso di ricordare che la selezione tra prima e seconda unità può avvenire attraverso una configurazione precisa, di solito con l'ausilio di ponticelli, oppure automatica (*cable select*), in cui la posizione del connettore nel cavo decide il numero dell'unità.

Di norma, le schede madri degli elaboratori conformi all'architettura IBM PC dispongono di due bus ATA, con cui si possono connettere complessivamente un massimo di quattro unità.

I comandi che si danno alle unità ATA comportano la scrittura e la lettura di «registri» interni alla gestione dei bus. Per accedere a tali registri, nell'architettura conforme al IBM PC, si usano delle porte di comunicazione: leggendo o scrivendo una certa porta, si ottiene la lettura o la scrittura di un certo registro del sistema ATA.

I registri sono sempre associati a un certo bus, sul quale però possono essere connessi due dispositivi. A seconda del contesto, i comandi che si impartiscono possono riguardare il bus in generale, o un dispositivo preciso, individuato dai parametri del comando.

Dal momento che le unità di memorizzazione di massa non possono essere sufficientemente veloci nelle loro reazioni, rispetto alle possibilità della CPU, ci sono alcune operazioni che richiedono un tempo di attesa prima di poter leggere un esito corretto o prima di poter proseguire con altri comandi. Nel documento *ATA PIO mode*, http://wiki.osdev.org/ATA_PIO_Mode, citato anche in fondo al capitolo, si menziona in certi casi un ritardo di sicurezza di 400 ns, necessario soprattutto per le unità più vecchie. In quel documento si fa riferimento alla possibilità di eseguire per cinque volte lo stesso comando, prima di poter ottenere un esito valido, ma questa procedura

riguarda la programmazione in linguaggio assembler, perché se si utilizza il C, o altro linguaggio più evoluto, può darsi che non ci sia la stessa efficienza e bastino meno tentativi.

Quando si eseguono operazioni di scrittura, occorre chiedere espressamente lo scarico della memoria trattenuta (*cache*), per ottenere la memorizzazione effettiva nel disco. Se non si ha l'accortezza di procedere in tal modo, si rischia di fare fallire l'operazione di scrittura successiva.

Anche le unità ATA, come tutti i sistemi di memorizzazione di massa a disco, possono trovarsi ad avere dei settori danneggiati e inutilizzabili. Nel caso delle unità ATA si distingue però tra settori che non possono essere letti o scritti in modo permanente e settori che invece non possono essere letti, ma solo temporaneamente. Questa impossibilità temporanea di lettura può derivare da una fase di scrittura incompleta: tali settori tornano a essere leggibili correttamente quando si esegue su di loro una nuova operazione di scrittura che giunge a termine correttamente.

83.9.1 Modalità di accesso

L'accesso ai settori delle unità ATA può avvenire secondo tre modalità: CHS, LBA28 e LBA48. La modalità CHS rappresenta il metodo più vecchio per individuare un settore in un disco, in quanto occorre specificare le coordinate composte da cilindro, testina e settore di questa combinazione. L'accesso in modalità CHS (*Cylinder Head Sector*) riguarda concretamente solo le unità a disco degli anni 1980, perché successivamente le unità ATA hanno introdotto la possibilità di raggiungere i settori attraverso un numero sequenziale, senza dover conoscere la geometria effettiva del disco.



Per problemi di compatibilità, è rimasta la facoltà di individuare i settori attraverso coordinate CHS, le quali di norma si riferiscono a una geometria astratta e non reale. Infatti, in condizioni normali, ci possono essere unità a disco composte da una quantità limitatissima di testine (due, quattro, sei), mentre programmi come **'fdisk'** riportano spesso una quantità fantastica di 255 testine. Tutto ciò deriva dai limiti del BIOS (*firmware*) degli elaboratori conformi all'architettura IBM PC.

AmMESSO di avere determinato o definito una certa geometria, si convertono le coordinate CHS in numero assoluto del settore con delle formule. Si considerano le variabili seguenti:

Variabile	Descrizione
<i>C</i>	Quantità totale dei cilindri.
<i>H</i>	Quantità totale delle testine.
<i>S</i>	Quantità totale di settori per traccia (ogni cilindro ha <i>H</i> tracce).
<i>c</i>	Cilindro di una coordinata, dove il primo è zero e l'ultimo è <i>C</i> -1.
<i>h</i>	Testina di una coordinata, dove la prima è zero e l'ultima è <i>H</i> -1.
<i>s</i>	Settore di una traccia, dove il primo è 1 e l'ultimo è <i>S</i> . Che i settori di una traccia inizino da uno è un fatto importante, a cui è necessario prestare attenzione.
<i>z</i>	Numero assoluto del settore, dove il primo è pari a zero.

Il numero assoluto di un settore, conoscendo la sua coordinata CHS, si ottiene come:

$$z = c \cdot H \cdot S + h \cdot S + s - 1$$

Partendo invece dal numero assoluto del settore, per determinare le

sue coordinate virtuali, valgono le formule seguenti. Si osservi che dalle divisioni si prendono solo i risultati interi.

$$c = \frac{z}{H \cdot S}$$

$$h = \frac{z + 1 - c \cdot H \cdot S}{S}$$

$$s = z + 1 - c \cdot H \cdot S - h \cdot S$$

Quando si accede alle unità ATA specificando il numero assoluto del settore, si può usare la modalità LBA28, che permette di raggiungere al massimo il settore FFFFFF_{16} , oppure, ammesso che il dispositivo lo consenta, la modalità LBA48, che permette di raggiungere al massimo il settore FFFFFFFFFFFF_{16} . In pratica, con la modalità LBA28, sapendo che i settori sono da 512 byte, si possono gestire dispositivi con una capacità massima di 128 Giabyte, mentre con la modalità LBA48 si può arrivare fino a 128 Piabyte ($128 \cdot 2^{50}$).

La modalità di accesso CHS è superata da molto tempo. Tuttavia, se la si deve usare, va ricordato che, in tal caso, non può esistere il settore zero.

83.9.2 Registri per la gestione delle unità ATA

Nella tabella successiva, si riepilogano i registri utili per la gestione delle unità ATA, secondo la modalità PIO. Nella tabella si omette *data port*, in quanto si riferisce soltanto alla modalità DMA per il trasferimento dei dati. Il registro che nella tabella è chiamato *data* è diverso dal *data port* e riguarda il trasferimento in modalità PIO.

I registri della tabella sono generalmente da un solo byte (8 bit), a eccezione di *data* il quale normalmente va letto e scritto a coppie di byte (16 bit).

I registri possono consentire la lettura (r), la scrittura (w) o entrambe le cose. Quando un registro è a senso unico (sola lettura o sola scrittura), vuol dire che l'accesso in senso opposto è relativo a informazioni differenti, a cui si associa un altro nome. Per esempio, quello che viene chiamato *regular status* è un registro in sola lettura, ma se vi si accede ugualmente in scrittura, si interviene in pratica nel registro *device control*, il quale è invece in sola scrittura (naturalmente lo stesso vale in senso inverso). Evidentemente l'attribuzione di nomi differenti ai registri, a seconda della direzione di accesso, consente di evitare facili confusioni.

Lo stato di funzionamento del dispositivo corrente di un certo bus è riportato in modo uguale da due registri: *regular status* e *alternate status*. La distinzione tra questi sta nel fatto che la lettura del primo comporta la «acquisizione» dell'informazione, mentre la lettura del secondo non altera in alcun modo la situazione del dispositivo. Lo stato del dispositivo è costituito da indicatori, ovvero bit che se sono a uno rappresentano l'avverarsi di una certa condizione. A questi indicatori si fa riferimento con un nome. I più importanti sono BSY (*busy*), DF (*drive fault*), DRQ (*data request*) e ERR (*error*).

Tabella 83.73. Registri ATA utili per la modalità PIO.

Definizione	Dir (r/w)	Bus ₀	Bus ₁	Bus ₂	Bus ₃	Descrizione
<i>data</i>	r/w	1F0 ₁₆	170 ₁₆	1E8 ₁₆	168 ₁₆	Consente di leggere o scrivere il dispositivo selezionato precedentemente con il registro <i>device</i> , a coppie di byte, ma prima di poterlo fare è necessario che l'indicatore DRQ sia attivo. La lettura o la scrittura è progressiva, ma riguarda sempre uno o più settori interi, pertanto va ripetuta a multipli di 256 volte. Non è possibile mescolare letture e scritture.
<i>error</i>	r/-	1F1 ₁₆	171 ₁₆	1E9 ₁₆	169 ₁₆	Questo registro, in sola lettura, descrive il tipo di errore manifestato dall'indicatore ERR. In pratica, il contenuto di questo registro è privo di significato se l'indicatore ERR non fosse attivo. Inoltre, per poter prendere in considerazione l'indicatore ERR, è opportuno che l'indicatore BSY sia a zero. L'interpretazione del registro cambia a seconda del tipo di comando che lo ha causato. L'accesso in scrittura a questo registro porta a intervenire invece nel registro <i>feature</i> .
<i>feature</i>	-/w	1F1 ₁₆	171 ₁₆	1E9 ₁₆	169 ₁₆	Registro in sola scrittura, per l'indicazione di un parametro, il cui significato cambia a seconda del comando che si impartisce successivamente. La scrittura in questo registro deve essere subordinata al fatto che gli indicatori BSY e DRQ siano a zero. L'accesso in lettura a questo registro porta a intervenire invece nel registro <i>error</i> .

Definizione	Dir (r/w)	Bus ₀	Bus ₁	Bus ₂	Bus ₃	Descrizione
<i>count</i> (<i>sector count</i>)	r/w	1F2 ₁₆	172 ₁₆	1EA ₁₆	16A ₁₆	Si usa come parametro di un comando successivo, per indicare una quantità di settori da prendere in considerazione, tenendo conto che il valore zero ha un significato particolare. Si può modificare il registro solo se gli indicatori BSY e DRQ sono a zero; allo stesso modo la lettura del registro è valida solo se questi indicatori sono a zero.
<i>low</i> (<i>lba low</i>)	r/w	1F3 ₁₆	173 ₁₆	1EB ₁₆	16B ₁₆	Si usa come parametro di un comando successivo, per indicare l'intervallo bit ₀ ..bit ₇ dell'indirizzo del settore da raggiungere. Si può modificare il registro solo se gli indicatori BSY e DRQ sono a zero; allo stesso modo la lettura del registro è valida solo se questi indicatori sono a zero.
<i>mid</i> (<i>lba mid</i>)	r/w	1F4 ₁₆	174 ₁₆	1EC ₁₆	16C ₁₆	Si usa come parametro di un comando successivo, per indicare l'intervallo bit ₈ ..bit ₁₅ dell'indirizzo del settore da raggiungere. Si può modificare il registro solo se gli indicatori BSY e DRQ sono a zero; allo stesso modo la lettura del registro è valida solo se questi indicatori sono a zero.
<i>high</i> (<i>lba high</i>)	r/w	1F5 ₁₆	175 ₁₆	1ED ₁₆	16D ₁₆	Si usa come parametro di un comando successivo, per indicare l'intervallo bit ₁₆ ..bit ₂₃ dell'indirizzo del settore da raggiungere. Si può modificare il registro solo se gli indicatori BSY e DRQ sono a zero; allo stesso modo la lettura del registro è valida solo se questi indicatori sono a zero.

Definizione	Dir (r/w)	Bus ₀	Bus ₁	Bus ₂	Bus ₃	Descrizione
<i>device</i>	r/w	1F6 ₁₆	176 ₁₆	1EE ₁₆	16E ₁₆	È un registro al cui interno si inseriscono diverse informazioni, il cui significato può cambiare a seconda del comando che si impartisce successivamente. Tuttavia, il bit ₄ (il quinto), rappresenta il dispositivo nel bus a cui si fa riferimento (zero è il primo, uno è il secondo). Non va confuso questo registro con il <i>device control</i> .
<i>regular status</i>	r/-	1F7 ₁₆	177 ₁₆	1EF ₁₆	16F ₁₆	Dà lo stato del bus, con una lettura che comporta l'acquisizione di tale informazione. Quando è attivo l'indicatore BSY, tutti gli altri indicatori sono privi di significato. L'interpretazione del dato è equivalente a quella che si deve fare per il registro <i>alternate status</i> . Questo registro è in sola lettura, in quanto, se vi si accede in scrittura, si interviene invece nel registro <i>command</i> .
<i>command</i>	-/w	1F7 ₁₆	177 ₁₆	1EF ₁₆	16F ₁₆	Consente di impartire un comando, sulla base di parametri già forniti attraverso altri registri. A eccezione del comando DEVICE RESET, in tutti gli altri casi si può scrivere in questo registro soltanto se gli indicatori BSY e DRQ sono entrambi a zero. Questo registro è in sola scrittura, in quanto, se vi si accede in lettura, si ottiene il contenuto del registro <i>regular status</i> .

Definizione	Dir (r/w)	Bus ₀	Bus ₁	Bus ₂	Bus ₃	Descrizione
<i>alternate status</i>	r/-	3F6 ₁₆	376 ₁₆	3E6 ₁₆	366 ₁₆	Dà lo stato del bus, attraverso una lettura «neutra», ovvero inerte. Quando è attivo l'indicatore BSY, tutti gli altri indicatori sono privi di significato. L'interpretazione del dato è equivalente a quella che si deve fare per il registro <i>regular status</i> . Questo registro è in sola lettura, in quanto, se vi si accede in scrittura, si interviene invece nel registro <i>device control</i> .
<i>control (device control)</i>	-/w	3F6 ₁₆	376 ₁₆	3E6 ₁₆	366 ₁₆	Consente di impostare alcuni indicatori che controllano la gestione dei dispositivi. Questo registro è in sola scrittura, in quanto, se vi si accede in lettura, si ottiene il contenuto del registro <i>alternate status</i> .

Nella tabella appena apparsa, sono indicati gli indirizzi di I/O per accedere a quattro diversi bus ATA, negli elaboratori che si rifanno all'architettura IBM PC. Va osservato che di norma sono disponibili solo due di tali bus (per un massimo di quattro dispositivi connessi complessivamente), pertanto, in tal caso vanno considerati solo i primi due di questi indirizzi.

Tabella 83.74. Indicatori dei registri di stato: *regular status* e *alternate status*.

bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
BSY <i>busy</i>	DR- DY <i>device ready</i>	DF <i>device fault</i>	--	DRQ <i>data reque- st</i>	--	--	ERR <i>error</i>

Tabella 83.75. Descrizione degli indicatori dei registri di stato: *regular status* e *alternate status*.

Indicatore	Descrizione
BSY <i>busy</i>	Indica che il dispositivo corrente del bus selezionato è impegnato e non si possono impartire dei comandi. Quando questo indicatore è attivo, gli altri, a parte ERR, sono privi di significato.
DRDY <i>device ready</i>	Questo indicatore non è l'opposto di BSY. Semplificando le cose, è a zero se il dispositivo è in pausa (motore fermo), mentre dovrebbe essere a uno in quasi tutte le altre situazioni.
DF <i>device fault</i>	Indica il verificarsi di un errore del dispositivo, diverso da quelli considerati dall'indicatore ERR.
DRQ <i>data request</i>	Indica che il dispositivo corrente è pronto per un trasferimento di dati. Al termine di un trasferimento, l'indicatore si azzera.
ERR <i>error</i>	Indica un errore, interpretabile leggendo il registro <i>error</i> .

Tabella 83.76. Bit del registro *device control*.

bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
HOB <i>high order byte</i>	--	--	--	--	SRST <i>software reset</i>	nIEN <i>not interrupt enabled</i>	0

Tabella 83.77. Descrizione dei bit del registro *device control*.

Bit	Descrizione
HOB <i>high order byte</i>	Riguarda una situazione specifica dell'accesso LBA 48; in tutti gli altri casi va lasciato a zero.
SRST <i>software reset</i>	Richiede l'inizializzazione software dei dispositivi connessi al bus.
nIEN <i>not interrupt enabled</i>	Richiede la disabilitazione dell'emissione di segnali di interruzioni da parte del dispositivo corrente.
bit ₀	Il bit meno significativo deve sempre essere lasciato a zero.

83.9.3 Alcuni comandi

«

L'invio di un comando al dispositivo corrente comporta l'indicazione di alcuni parametri utilizzando i registri, tra cui, soprattutto, il codice del comando stesso. Il comando viene eseguito nel momento in cui si scrive nel registro *command* il codice che lo identifica, pertanto questa scrittura va fatta per ultima.

Se prima di dare un comando si intende agire anche sul registro *device control*, per esempio per inibire l'emissione di interruzioni

per il dispositivo coinvolto, la scrittura in tale registro deve avvenire prima della scrittura nel registro *command* (per ovvi motivi), ma successivamente alla selezione del dispositivo con la scrittura nel registro *device*.

83.9.3.1 Comando READ SECTORS

Il comando READ SECTORS, corrispondente al codice 20_{16} , consente di leggere uno o più settori, in modalità PIO, dal dispositivo specificato nel registro *device*, con indirizzamento LBA28.

Tabella 83.78. Parametri del comando «READ SECTORS».

Registro	bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
<i>device</i>	--	LBA	--	DEV	Bit bit ₂₄ ..bit ₂₈ dell'indirizzo del primo settore da leggere.			
<i>feature</i>	--							
<i>sector count</i>	Quantità di settori da leggere.							
<i>LBA low</i>	Bit bit ₀ ..bit ₇ dell'indirizzo del primo settore da leggere.							
<i>LBA mid</i>	Bit bit ₈ ..bit ₁₅ dell'indirizzo del primo settore da leggere.							
<i>LBA high</i>	Bit bit ₁₆ ..bit ₂₃ dell'indirizzo del primo settore da leggere.							
<i>command</i>	20_{16}							

Tabella 83.79. Descrizione dei registri coinvolti.

Registro	Utilizzo
<i>feature</i>	Non viene considerato dal comando.

Registro	Utilizzo
<i>device</i>	Il bit ₅ e il bit ₇ possono essere impostati a uno per mantenere la compatibilità con vecchi dispositivi, ma in generale sono semplicemente ignorati. Il bit ₆ , se attivo, indica che il primo settore da leggere viene individuato come numero assoluto (LBA), mentre se viene lasciato a zero, significa che tale settore viene raggiunto con coordinate CHS (cilindro, testina, settore). Il bit ₄ individua il dispositivo, distinguendo tra primo (0) e secondo (1) del bus a cui si fa riferimento. L'intervallo di bit ₀ ..bit ₃ si utilizza per annotare la parte più significativa di un indirizzo LBA28, per raggiungere il primo settore da leggere.
<i>lba low</i> <i>lba mid</i> <i>lba high</i>	Questi tre registri rappresentano complessivamente i primi 24 bit dell'indirizzo del primo settore da raggiungere.

Dopo l'invio del comando si deve attendere che l'indicatore BSY torni a zero e che l'indicatore DRQ si attivi, per poi procedere alla lettura di *data*, generalmente a coppie di byte, fino al completamento della dimensione dei settori richiesti, quando l'indicatore DRQ torna a zero.

Tabella 83.80. Esito normale, atteso dopo l'esecuzione completa del comando, nel registro di stato.

bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
BSY	DR-DY	DF	--	DRQ	--	--	ERR
0	1	0	--	0	--	--	0

83.9.3.2 Comando WRITE SECTORS

Il comando WRITE SECTORS, corrispondente al codice 30_{16} , consente di scrivere uno o più settori, in modalità PIO, nel dispositivo specificato nel registro *device*, con un indirizzamento LBA28.

Tabella 83.81. Parametri del comando WRITE SECTORS.

Registro	bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
<i>device</i>	--	LBA	--	DEV	Bit bit ₂₄ ..bit ₂₈ dell'indirizzo del primo settore da scrivere.			
<i>feature</i>	--							
<i>sector count</i>	Quantità di settori da scrivere.							
<i>LBA low</i>	Bit bit ₀ ..bit ₇ dell'indirizzo del primo settore da scrivere.							
<i>LBA mid</i>	Bit bit ₈ ..bit ₁₅ dell'indirizzo del primo settore da scrivere.							
<i>LBA high</i>	Bit bit ₁₆ ..bit ₂₃ dell'indirizzo del primo settore da scrivere.							
<i>command</i>	30_{16}							

Tabella 83.82. Descrizione dei registri coinvolti.

Registro	Utilizzo
<i>feature</i>	Non viene considerato dal comando.

Registro	Utilizzo
<i>device</i>	Il bit ₅ e il bit ₇ possono essere impostati a uno per mantenere la compatibilità con vecchi dispositivi, ma in generale sono semplicemente ignorati. Il bit ₆ , se attivo, indica che il primo settore da leggere viene individuato come numero assoluto (LBA), mentre se viene lasciato a zero, significa che tale settore viene raggiunto con coordinate CHS (cilindro, testina, settore). Il bit ₄ individua il dispositivo, distinguendo tra primo (0) e secondo (1) del bus a cui si fa riferimento. L'intervallo di bit ₀ ..bit ₃ si utilizza per annotare la parte più significativa di un indirizzo LBA, per raggiungere il primo settore da scrivere.
<i>lba low</i> <i>lba mid</i> <i>lba high</i>	Questi tre registri rappresentano complessivamente i primi 24 bit dell'indirizzo del primo settore da raggiungere.

Dopo l'invio del comando si deve attendere che l'indicatore BSY torni a zero e che l'indicatore DRQ si attivi, per poi procedere alla scrittura di *data*, generalmente a coppie di byte, fino al completamento della dimensione dei settori richiesti, quando l'indicatore DRQ torna a zero.

Tabella 83.83. Esito normale, atteso dopo l'esecuzione completa del comando, nel registro di stato.

bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
BSY	DR-DY	DF	--	DRQ	--	--	ERR
0	1	0	--	0	--	--	0

83.9.3.3 Comando FLUSH CACHE

Il comando CACHE FLUSH, corrispondente al codice $E7_{16}$, assicura la memorizzazione dei settori modificati ed è necessario inviarlo prima di procedere con ulteriori comandi di scrittura. L'operazione riguarda il dispositivo specificato nel registro *device*.

Tabella 83.84. Parametri del comando FLUSH CACHE.

Registro	bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
<i>device</i>	--	--	--	DEV	--			
<i>feature</i>	--							
<i>sector count</i>	--							
<i>LBA low</i>	--							
<i>LBA mid</i>	--							
<i>LBA high</i>	--							
<i>command</i>	$E7_{16}$							

Tabella 83.85. Esito normale, atteso dopo l'esecuzione completa del comando, nel registro di stato.

bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
BSY	DR-DY	DF	--	DRQ	--	--	ERR
0	1	0	--	0	--	--	0

83.9.3.4 Comando IDENTIFY DEVICE

Il comando IDENTIFY DEVICE, corrispondente al codice EC_{16} , consente di interrogare le caratteristiche di un dispositivo ATA, esclusi i dispositivi ATAPI.

Tabella 83.86. Parametri del comando IDENTIFY DEVICE.

Registro	bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
<i>device</i>	--	--	--	DEV	--			
<i>feature</i>	--							
<i>sector count</i>	--							
<i>LBA low</i>	--							
<i>LBA mid</i>	--							
<i>LBA high</i>	--							
<i>command</i>	EC ₁₆							

Dopo l'invio del comando, si deve verificare il contenuto di uno dei due registri di stato: se questo fosse a zero, significa che il dispositivo richiesto non esiste. Se invece il registro contiene qualcosa, si deve attendere che l'indicatore BSY torni a zero e che l'indicatore DRQ si attivi, per poi procedere alla lettura di *data*, a blocchi da 16 bit, per 256 volte (in totale si hanno 512 byte, come un settore comune), quando l'indicatore DRQ torna a zero. All'interno di questi blocchi da 16 bit ci sono informazioni che consentono di conoscere nel dettaglio le caratteristiche del dispositivo.

Se invece di un indicatore DRQ attivo si ottiene un errore, rappresentato quindi dall'indicatore ERR attivo, vanno letti i registri *lba mid* e *lba high*:

Tabella 83.87. Contenuto dei registri *lba mid* e *lba high* in caso di errore ottenuto a seguito del comando IDENTIFY DEVICE.

Tipo di errore	Registro <i>lba mid</i>	Registro <i>lba high</i>
Si tratta di unità ATA, ma si è verificato ugualmente un errore.	00 ₁₆	00 ₁₆
Si tratta di un'unità ATAPI parallela.	14 ₁₆	EB ₁₆
Si tratta di un'unità SATA.	3C ₁₆	C3 ₁₆
Si tratta di un'unità ATAPI seriale.	69 ₁₆	96 ₁₆

Tabella 83.88. Esito normale, atteso dopo l'esecuzione completa del comando, inclusa la lettura del registro *data*, nel registro di stato.

bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
BSY	DR-DY	DF	--	DRQ	--	--	ERR
0	1	0	--	0	--	--	0

Tabella 83.89. Alcuni dati significativi ottenuti dalla lettura del registro *data*, dopo il comando IDENTIFY DEVICE. I blocchi da 16 bit letti sono numerati da 0 (il primo), fino a 255 (l'ultimo).

Blocco	Contenuto
60, 61	Complessivamente formano un numero a 32 bit che rappresenta la quantità totale di settori che si possono indirizzare in modalità LBA28. Se questo valore fosse a zero, indicherebbe che il dispositivo non è in grado di utilizzare un indirizzamento LBA28.
100, 101, 102, 103	Complessivamente formano un numero a 64 bit che rappresenta la quantità totale di settori che si possono indirizzare in modalità LBA48. Se questo valore fosse a zero, indicherebbe che il dispositivo non è in grado di utilizzare un indirizzamento LBA48, ma non è detto il contrario.

83.9.4 Individuazione dei bus utilizzati

«

Il tentativo di comunicare con un bus privo di unità collegate, comporta delle risposte errate, pertanto è necessario, prima di ogni altra cosa, scandire i bus presenti per verificare quali di questi sono effettivamente utilizzabili. Si tratta di leggere il contenuto del registro di stato (il *regular status* per la precisione): se questo ha tutti i bit a uno, si tratta di un bus a cui non è collegato alcunché.

Il pezzetto di codice seguente, attraverso la funzione *inb()*, che si intuisce serva a leggere un byte da una porta di I/O, si ottiene lo stato del primo bus, corrispondente all'indirizzo $1F7_{16}$.

```

...
status = inb (0x1F7);
if (status == 0xFF)
{
    // Non ci sono dispositivi nel primo bus.
    ...
}
else
{
    // Ci potrebbe essere almeno un dispositivo nel primo
    // bus.
    ...
}
...

```

83.9.5 Azzeramento dello stato dei dispositivi

Quando si verifica un errore, le unità ATA normali (non ATAPI) richiedono un azzeramento software, provocato attraverso il bit SRST attivo nel registro *device control*. L'azzeramento riguarda però tutti i dispositivi connessi al bus, e, d'altro canto, non essendo coinvolto in questo un comando, non ci sarebbe il modo di precisare un dispositivo particolare. Va osservato che una volta scritto nel registro *device control* il valore corrispondente all'attivazione del bit SRST, occorre riscrivere un valore pari a zero per questo bit, altrimenti il bus rimarrebbe in uno stato di inizializzazione.

```

...
outb (0x3F6, 0x02);    // Azzerà.
outb (0x3F6, 0x00);    // Ripristina la condizione normale.
...

```

83.9.6 Controllo delle interruzioni

«

In varie occasioni, i dispositivi possono mettersi in uno «stato di interruzione», a cui corrisponde effettivamente un'interruzione hardware nell'architettura IBM PC. Dal momento che le situazioni in cui tali interruzioni si verificano sono varie e complesse, la loro gestione potrebbe essere troppo impegnativa. D'altro canto è possibile gestire i dispositivi ATA anche senza considerare le interruzioni.

A questo proposito è possibile scrivere nel registro *device control* il valore 01_{16} , corrispondente al bit nIEN attivo, ogni volta che è appena stato selezionato un dispositivo nel registro *device*, per evitare che il comando che si va a impartire produca poi un'interruzione.

83.9.7 Verifica dell'esito di un comando

«

Ogni volta che si dà un comando a un dispositivo ATA, se non si vogliono considerare le interruzioni, occorre controllare ripetutamente il registro di stato, precisamente il *regular status*, per sapere quando è possibile procedere ulteriormente.

Si deve attendere che l'indicatore BSY si azzeri, quindi si deve verificare che gli indicatori ERR e DF siano a zero: se uno dei due ha un valore diverso, significa che si è verificato un errore. Se gli indicatori di errore sono a zero, se dopo il comando ci si attende di leggere o scrivere dati attraverso il registro *data*, prima di poterlo fare, è necessario che l'indicatore DRQ sia attivo. Nell'esempio successivo si interroga l'esito di un comando appena impartito a un dispositivo del primo bus:

...

```
// Legge 8 bit dal registro «regular status».
```

```

status = inb (0x1F7);
while (status & 0x80)
{
    // BSY: continua a interrogare fino a che
    //      l'indicatore si azzerava.
    status = inb (0x1F7);
}
if (status & 0x21)
{
    // DF o ERR.
    return ...;
}
if (status & 0x08)
{
    // DRQ
    for (i = 0; i < 256; i++)
    {
        // Trasferisce dati attraverso il registro
        // «data».
        ...
    }
}
...

```

83.9.8 Identificazione delle unità

Una volta chiarito quali sono i bus che potrebbero contenere almeno un dispositivo, per sapere quali dispositivi sono presenti effettivamente e per conoscere le caratteristiche delle unità ATA presenti, si deve utilizzare il comando IDENTIFY DEVICE. A titolo di esempio si propone una funzione semplificata che riceve l'indicazione del numero del bus e del dispositivo di cui si vuole conoscere la dimensione massima in settori (da 512 byte), per un accesso in modalità



LBA28: se la funzione restituisce zero, significa che il dispositivo non è disponibile o non può operare in modalità LBA28 oppure si è verificato un errore che ne impedisce l'identificazione. Nell'esempio, la funzione *outb()* serve a scrivere un byte in una certa porta di I/O, mentre la funzione *inw()* serve a leggere un intero a 16 bit da una certa porta.

```
unsigned int
identify_device (int bus, int drive)
{
    int          reg_device;
    int          reg_command;
    int          reg_status;
    int          reg_data;
    int          reg_control;
    unsigned char device;
    unsigned char status;
    int          i;
    uint16_t     id[256];
    unsigned int size;
    //
    switch (bus)
    {
        case 0:
            reg_device = 0x1F6;
            reg_command = 0x1F7;
            reg_status = 0x1F7;
            reg_data = 0x1F0;
            reg_control = 0x3F6;
            break;
        case 1:
            ...
            break;
        ...
    }
```

```
    }
    device = 0x00;
    if (drive)
    {
        device = 0x10;
    }
    // Scrive 8 bit nel registro «device».
    outb (reg_device, device);
    // Scrive 8 bit nel registro «control».
    outb (reg_control, 0x01);
    // Scrive 8 bit nel registro «command».
    outb (reg_command, 0xEC);
    // Legge 8 bit dal registro «regular status».
    status = inb (reg_status);
    if (status == 0)
    {
        // L'unità non c'è.
        return 0;
    }
    while (status & 0x80)
    {
        // BSY
        status = inb (reg_status);
    }
    if (status & 0x21)
    {
        // DF o ERR: potrebbe trattarsi di unità ATAPI o
        // SATA.
        return 0;
    }
    //
    if (status & 0x08)
    {
        // DRQ
```

```
    for (i = 0; i < 256; i++)
    {
        // Legge 16 bit dal registro «data».
        id[i] = inw (reg_data);
    }
}
else
{
    // Per un motivo inspiegabile, non si ottiene
    // il permesso di leggere i dati.
    return 0;
}
// Si estrapola la quantità di settori disponibili in
// modalità LBA28: si prende il contenuto della memoria,
// partendo da 'id[60]', per un'estensione di 32 bit
// (4 byte), che così include anche 'id[61]',
// trasformando la cosa attraverso dei puntatori.
// Il meccanismo funziona perché i valori si
// intendono rappresentati in modalità «little endian»,
// per cui i byte meno significativi del valore appaiono
// per primi.
size = *((uint32_t *) &(id[60]));
//
return size;
}
```

83.9.9 Scomposizione dell'indirizzo

«

Quando si utilizzano comandi di lettura e scrittura di uno o più settori, si deve specificare l'indirizzo di questo, suddiviso in qualche modo nei registri che rappresentano i parametri del comando. Si distinguono tre casi, in base alle tre modalità di accesso: CHS, LBA28 e LBA48.

Tabella 83.94. Collocazione delle coordinate CHS.

Registro	bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
<i>device</i>	--	0	--	DEV	h (testina)			
<i>sector count</i>	quantità di settori da trattare							
<i>LBA low</i>	s (numero del settore della traccia, a partire da uno)							
<i>LBA mid</i>	c (cilindro) bit ₀ ..bit ₇							
<i>LBA high</i>	c (cilindro) bit ₈ ..bit ₁₅							

Tabella 83.95. Collocazione dell'indirizzo LBA28.

Registro	bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
<i>device</i>	--	1	--	DEV	Bit bit ₂₄ ..bit ₂₈			
<i>sector count</i>	quantità di settori da trattare							
<i>LBA low</i>	Bit bit ₀ ..bit ₇							
<i>LBA mid</i>	Bit bit ₈ ..bit ₁₅							
<i>LBA high</i>	Bit bit ₁₆ ..bit ₂₃							

Le due tabelle già apparse mostrano come articolare l'informazione CHS o l'indirizzo LBA28, assieme alla quantità di settori da prendere in considerazione. Nel caso in cui fosse specificata una quantità di settori pari a zero, si intenderebbero invece 256. Per la modalità LBA48, si procede in modo simile alla LBA28, con la differenza che i registri vanno scritti in due tornate e che il registro *device* non contiene alcuna porzione di questo.

Tabella 83.96. Utilizzo dei registri per collocare un indirizzo LBA48 in due fasi.

Registro	bit ₇	bit ₆	bit ₅	bit ₄	bit ₃	bit ₂	bit ₁	bit ₀
<i>device</i>	--	1	--	DEV	--			
<i>sector count</i>	quantità di settori bit ₈ ..bit ₁₅							
<i>LBA low</i>	Bit bit ₂₄ ..bit ₃₁							
<i>LBA mid</i>	Bit bit ₃₂ ..bit ₃₉							
<i>LBA high</i>	Bit bit ₄₀ ..bit ₄₈							
<i>sector count</i>	quantità di settori bit ₀ ..bit ₇							
<i>LBA low</i>	Bit bit ₀ ..bit ₇							
<i>LBA mid</i>	Bit bit ₈ ..bit ₁₅							
<i>LBA high</i>	Bit bit ₁₆ ..bit ₂₃							

In modalità LBA48, se si indica una quantità di settori pari a zero, si intendono invece 65536 settori.

83.9.10 Lettura LBA28 PIO

«

Viene proposto un esempio di funzione per la lettura di un settore, fornendo il numero del bus, del dispositivo all'interno del bus, il numero del settore (partendo da zero) e il puntatore all'inizio della memoria tampone che deve ricevere il settore. La funzione richiede ancora la verifica dei dati in ingresso e manca la possibilità di far scadere il ciclo di lettura del registro di stato nel caso in cui passasse troppo tempo.

La scrittura del registro *device* avviene per prima, per individuare

subito il dispositivo e per consentire la scrittura successiva del registro *control*, allo scopo di inibire una risposta tramite segnale di interruzione. Per il resto tutto procede come richiesto per il comando **READ SECTORS**.

```
int
read_sector (int bus, int drive, unsigned int sector,
             void *buffer)
{
    int         reg_device;
    int         reg_control;
    int         reg_feature;
    int         reg_count;
    int         reg_lba_low;
    int         reg_lba_mid;
    int         reg_lba_high;
    int         reg_command;
    int         reg_status;
    int         reg_data;
    unsigned char device;
    unsigned char lba_low;
    unsigned char lba_mid;
    unsigned char lba_high;
    unsigned char status;
    int         i;
    uint16_t    *destination = (uint16_t *) buffer;
    //
    switch (bus)
    {
        case 0:
            reg_device    = 0x1F6;
            reg_control   = 0x3F6;
            reg_feature   = 0x1F1;
            reg_count     = 0x1F2;
            reg_lba_low   = 0x1F3;
```

```
        reg_lba_mid   = 0x1F4;
        reg_lba_high  = 0x1F5;
        reg_command   = 0x1F7;
        reg_status    = 0x1F7;
        reg_data      = 0x1F0;
        break;
    case 1:
        ...
        break;
    ...
}
...
// Preparazione e scrittura del registro «device». Nel
// registro va specificato il fatto che si utilizza un
// accesso LBA, il dispositivo e la parte più
// significativa dell'indirizzo LBA28.
device = 0x00;
device |= 0x40;                // LBA.
if (drive)
{
    device |= 0x10;           // Secondo dispositivo;
}
device |= ((sector & 0x0F000000) >> 24);
outb (reg_device, device);
// Registro «control», per disabilitare il segnale di
// interruzione.
outb (reg_control, 0x02);
// Registro «feature».
outb (reg_feature, 0);
// Registro «sector count»: si vuole leggere un solo
// settore.
outb (reg_count, 1);
// LBA low, mid e high.
lba_low = (sector & 0x000000FF);
```

```
lba_mid = ((sector & 0x0000FF00) >> 8);
lba_high = ((sector & 0x00FF0000) >> 16);
outb (reg_lba_low, lba_low);
outb (reg_lba_mid, lba_mid);
outb (reg_lba_high, lba_high);
// Registro «command».
outb (reg_command, 0x20);
// Si attende che lo stato del dispositivo corrente
// torni a essere pronto.
status = inb (reg_status);
while (status & 0x80)
{
    // BSY
    status = inb (reg_status);
}
if (status & 0x21)
{
    // DF o ERR.
    return -1;
}
//
if (status & 0x08)
{
    // DRQ: si procede alla lettura del settore.
    for (i = 0; i < 256; i++)
    {
        // Legge 16 bit dal registro «data».
        destination[i] = inw (reg_data);
    }
}
else
{
    // Errore sconosciuto.
    return -1;
}
```

```

    }
    // Attesa ulteriore che l'unità sia di nuovo pronta.
    status = inb (reg_status);
    while (status & 0x80)
    {
        // BSY
        status = inb (reg_status);
    }
    if (status & 0x21)
    {
        // DF o ERR.
        return -1;
    }
    // Fine normale.
    return 0;
}

```

83.9.11 Scrittura LBA28 PIO

«

Viene proposto un esempio di funzione per la scrittura di un settore, fornendo il numero del bus, del dispositivo all'interno del bus, il numero del settore (partendo da zero) e il puntatore all'inizio della memoria tampone che contiene il settore da scrivere. La funzione è molto simile a quella proposta per la lettura e vengono omesse le porzioni uguali.

La differenza fondamentale sta nel fatto che, dopo la scrittura del settore (con l'aiuto della funzione *outw()*), dopo aver verificato che il registro di stato segnala la conclusione corretta dell'operazione, va dato un comando di scarico della memoria trattenuta (*cache*).

```

int
write_sector (int bus, int drive, unsigned int sector,

```

```
        void *buffer)
{
    ...
    uint16_t      *source = (uint16_t *) buffer;
    ...
    ...
    // Registro «command».
    outb (reg_command, 0x30);
    ...
    if (status & 0x08)
    {
        // DRQ: si procede alla scrittura del settore.
        for (i = 0; i < 256; i++)
        {
            // Scrive 16 bit nel registro «data».
            outw (reg_data, source[i]);
        }
    }
    ...
    // Quando il dispositivo torna a essere pronto, si dà
    // un comando di scarico della memoria trattenuta:
    // CACHE FLUSH. Il registro «device» contiene già
    // tutti i dati che servono e non c'è bisogno di
    // aggiornarlo.
    outb (reg_command, 0xE7);
    // Si attende ancora che il dispositivo torni a essere
    // pronto, prima di concludere.
    status = inb (reg_status);
    while (status & 0x80)
    {
        // BSY
        status = inb (reg_status);
    }
    if (status & 0x21)
```

```
{
    // DF o ERR.
    return -1;
}
// Fine normale.
return 0;
}
```

83.10 PCI

«

Il bus PCI (*Peripheral component interconnect*) si occupa di organizzare le risorse assegnate ai componenti a lui connessi, offrendo al programmatore le informazioni necessarie per accedervi. Tale organizzazione automatica avviene attraverso una procedura software, la quale però è già inclusa nel firmware degli elaboratori IBM AT.

Su un bus PCI possono essere connessi altri bus, sia per estenderlo, sia per permettere il collegamento con altri tipi di bus. Quando si deve accedere a un componente che si trova in un bus secondario, si creano delle complicazioni che qui non vengono considerate, limitando l'attenzione al caso in cui si gestiscano esclusivamente componenti PCI connessi direttamente al primo bus. Pertanto, con questa semplificazione, è sufficiente interrogare i registri del bus PCI per sapere quali dispositivi sono connessi e quali riferimenti servono per comunicare con loro.

Va tenuto in considerazione che tutta la gestione del bus PCI prevede che i dati contenuti in scalari interi siano organizzati in modalità *little endian*; pertanto, in un'architettura x86 non si creano problemi di conversione, ammesso che si rispetti la struttura prevista per questi valori.

83.10.1 Registri e porte

Per comunicare con il bus PCI, allo scopo di ottenere le informazioni che servono sul bus e sui componenti inseriti, si usano due porte di I/O. La prima, corrispondente all'indirizzo $0CF8_{16}$, serve per fornire delle coordinate, con cui individuare un registro accessibile attraverso la seconda, corrispondente all'indirizzo $0CFC_{16}$.

Indirizzo I/O	Accesso	Nome mnemonico	Descrizione
$0CF8_{16}$	32 bit	<i>address</i>	Scrivendo un valore a 32 bit, strutturato opportunamente, si individua un registro accessibile poi all'indirizzo <i>data</i> .
$0CFC_{16}$	32 bit	<i>data</i>	Si legge o si scrive un valore a 32 bit che costituisce l'informazione a cui puntano le coordinate fornite in precedenza attraverso l'indirizzo <i>address</i> .

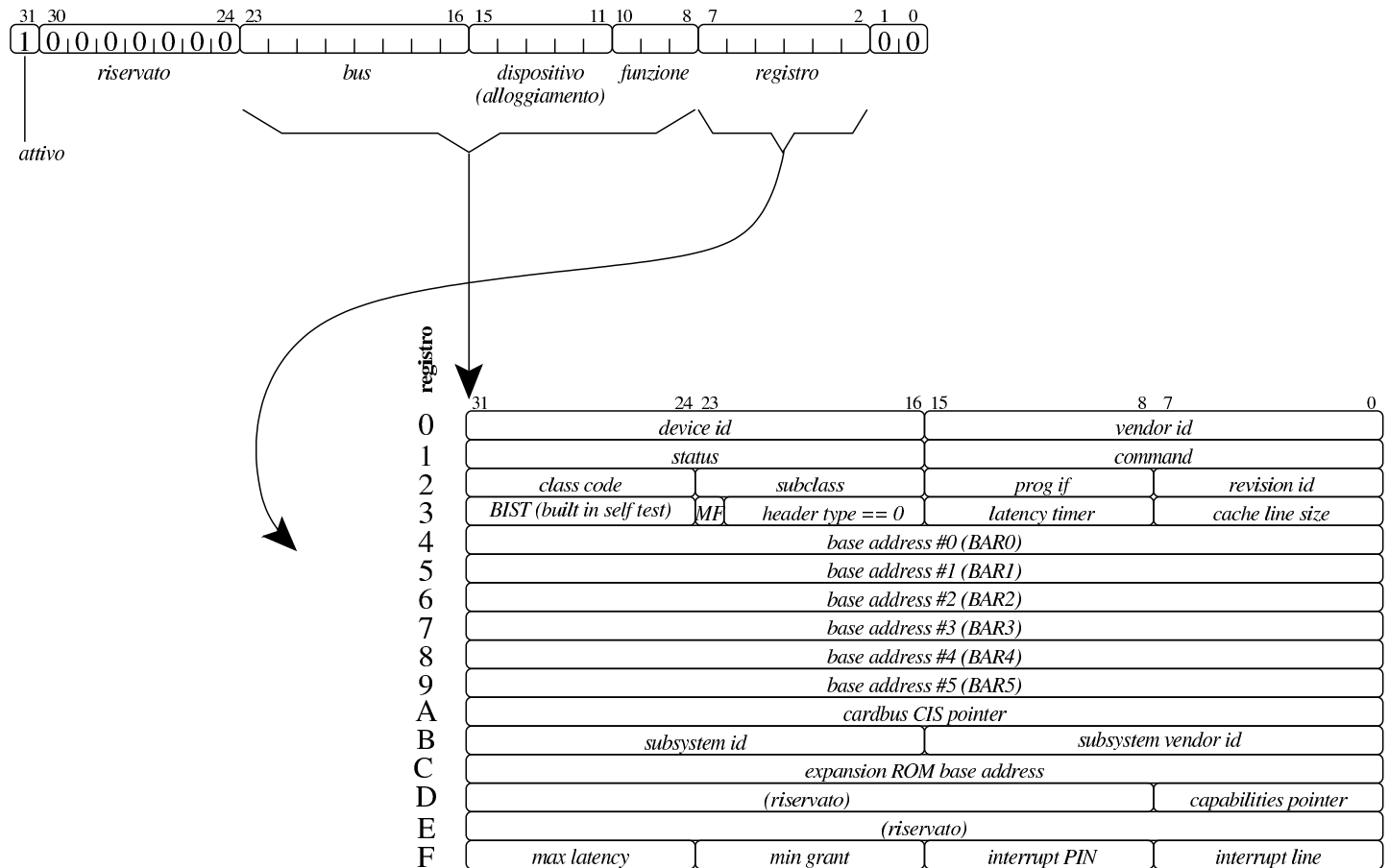
Le informazioni che il bus PCI ha da offrire sono strutturate in «tabelle» (ma nella documentazione standard si parla di «intestazioni»), suddivise in registri da 32 bit, suddivisi a loro volta in campi non omogenei tra loro. L'accesso a queste tabelle avviene registro per registro, attraverso la porta $0CFC_{16}$ (*data*), ma per individuare il registro a cui si è interessati, prima di ciò occorre scrivere un valore appropriato nella porta $0CF8_{16}$ (*address*).

Per raggiungere un dato occorre conoscere il numero del bus, dell'alloggiamento e del registro.

83.10.2 Strutture dei dati

«

La figura successiva mostra le strutture in cui si articolano il selettore per individuare un registro e la tabella dei registri corrispondente (viene mostrata solo la tabella di tipo 00_{16}).

Figura 83.100. Selettore e tabella di tipo 00_{16} .

Vengono proposte anche due strutture, in linguaggio C, per descrivere il selettore e la tabella di tipo 00_{16} .

Listato 83.101. Struttura del selettore. Il membro *selector* consente di azzerare velocemente tutto il selettore, senza dover intervenire sui singoli campi.

```
typedef union {
    uint32_t    selector;
    struct {
        uint32_t zero      : 2,
                reg       : 6,
                function  : 3,
                slot      : 5,
                bus       : 8,
                reserved  : 7,
                enable    : 1;
    };
} pci_address_t;
```

Listato 83.102. Struttura della tabella di tipo 00_{16} . L'array *r[16]* consente di individuare facilmente un registro nel suo complesso.

```
typedef union {
    uint32_t r[16];
    struct {
        struct {
            uint32_t vendor_id          : 16,
                    device_id         : 16;

            //
            uint32_t command            : 16,
                    status             : 16;

            //
            uint32_t revision_id        : 8,
                    prog_if            : 8,
                    subclass           : 8,
                    class_code         : 8;

            //
            uint32_t cache_line_size   : 8,
```

```
        latency_timer           : 8,  
        header_type             : 7,  
        multi_function          : 1,  
        bist                     : 8;  
  
    //  
    uint32_t  bar0;  
    uint32_t  bar1;  
    uint32_t  bar2;  
    uint32_t  bar3;  
    uint32_t  bar4;  
    uint32_t  bar5;  
    uint32_t  cardbus_cis_pointer;  
    uint32_t  expansion_rom_base_address;  
    //  
    uint32_t  subsystem_vendor_id      : 16,  
             subsystem_id              : 16;  
  
    //  
    uint32_t  capabilities_pointer     : 8,  
             reserved_1                : 24;  
  
    //  
    uint32_t  reserved_2;  
    //  
    uint32_t  interrupt_line           : 8,  
             interrupt_pin             : 8,  
             min_grant                 : 8,  
             max_latency               : 8;  
};  
};  
} pci_header_type_00_t;
```

Tabella 83.103. Campi in cui si articola il selettore.

Bit	Denominazione	Descrizione
31	<i>enable</i>	Questo bit deve essere posto a uno.
30..24	<i>reserved</i>	Campo non utilizzato, da lasciare azzerato.
23..16	<i>bus</i>	Il numero del bus a cui si intende fare riferimento. Il primo bus è quello corrispondente al numero zero.
15..11	<i>device</i>	Il numero dell'alloggiamento all'interno del bus, a partire da zero.
10..8	<i>funzione</i>	Nel caso di un dispositivo che incorpora più funzioni (per esempio più interfacce di rete assieme), questo è il numero che consente di selezionare il componente singolo, ovvero la funzione singola, tra tutte quelle incorporate.
7..2 1..0	<i>register</i>	I bit 7..2 individuano il numero del registro relativo alla tabella (<i>header</i>) che raccoglie la configurazione del dispositivo selezionato. I primi due bit rimangono generalmente a zero, perché l'accesso alla tabella è a blocchi da 32 bit, ovvero 4 byte, pertanto non c'è motivo di raggiungere posizioni intermedie.

La tabella di tipo 00_{16} , a cui si fa riferimento qui, è quella che riguarda i dispositivi comuni. Hanno invece tabelle differenti i dispositivi che connettono assieme più bus, dello stesso tipo o di tipo differente. A ogni modo, per facilitare un po' le cose, i primi quattro registri di queste tabelle sono uguali in tutte le tipologie.

Tabella 83.104. Campi principali della tabella di tipo 00_{16} .

Regi- stro	Bit	Denominazione	Descrizione
0_{16}	31..16	<i>device id</i>	Codice identificativo del tipo di dispositivo, stabilito dal costruttore.
0_{16}	15..0	<i>vendor id</i>	Codice identificativo del costruttore: se in questo campo si ottiene il valore $FFFF_{16}$, significa che il dispositivo non è presente nel bus e nell'alloggiamento cercato.
2_{16}	31..24 23..16 15..8	<i>class code</i> <i>subclass</i> <i>prog if</i>	Codice che descrive la classe, la sottoclasse e la funzione del dispositivo (tabella 83.107).
3_{16}	23	<i>multi function</i>	Se questo bit è a uno, significa che il dispositivo è multifunzione.
3_{16}	22..16	<i>header type</i>	Codice che descrive la struttura della tabella: 00_{16} indica quella di un dispositivo normale, 01_{16} riguarda un ponte tra bus PCI e 02_{16} riguarda un ponte verso unità cardbus.

Regi- stro	Bit	Denominazione	Descrizione
$4_{16}..9_{16}$	31..0	<i>BAR0</i> <i>BAR1</i> <i>BAR2</i> <i>BAR3</i> <i>BAR4</i> <i>BAR5</i>	Si tratta di indirizzi che rappresentano un riferimento nella memoria o negli indirizzi di I/O. Con i dispositivi a cui si accede attraverso un indirizzo IRQ e un solo indirizzo I/O, BAR0 corrisponde all'indirizzo di I/O. Tuttavia, il valore contenuto in questi registri va interpretato e non si può usare tale e quale, come appare (figura 83.105).
F_{16}	7..0	<i>Interrupt line</i>	Si tratta del numero di IRQ usato dal dispositivo, ma se si ottiene il valore FF_{16} , vuol dire che il dispositivo non utilizza un IRQ.

Figura 83.105. Interpretazione di un indirizzo contenuto in un campo BAR_n . Si osserva che il bit meno significativo consente di capire se si tratta di un indirizzo in memoria o di una porta di I/O.

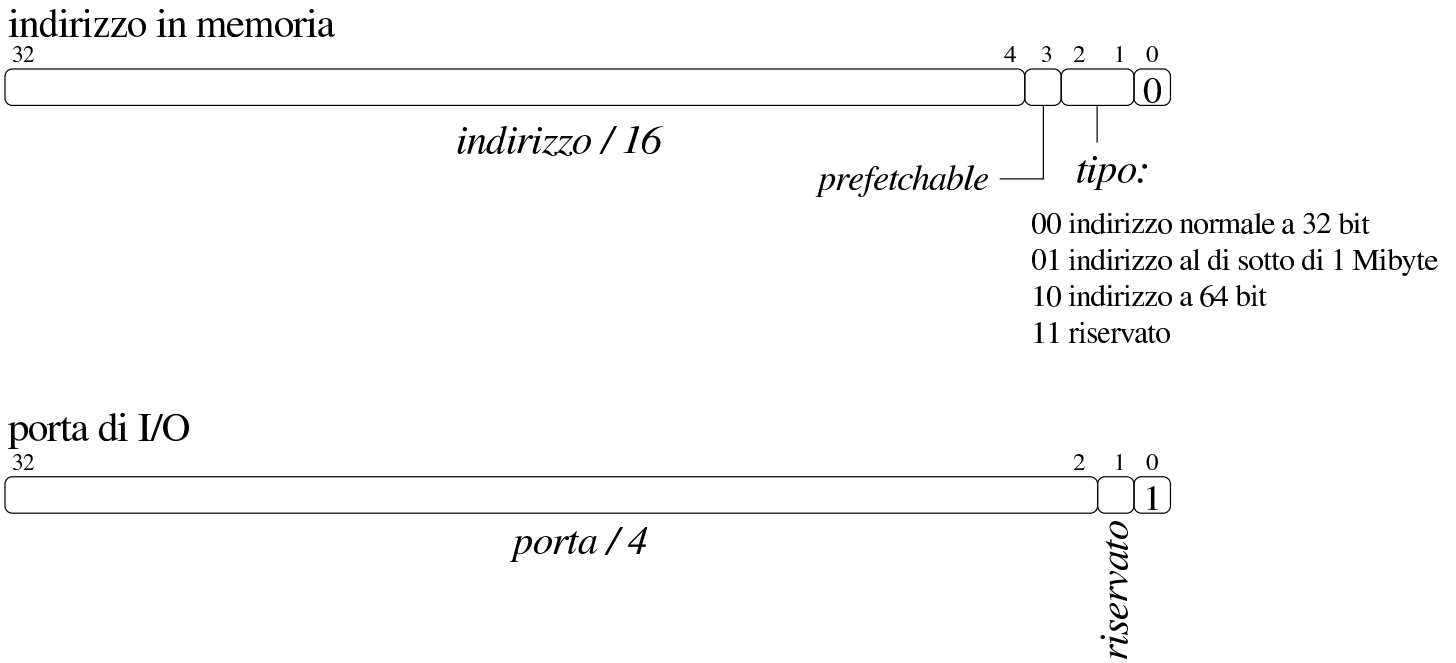


Tabella 83.106. Interpretazione della classe del dispositivo (*class code*).

<i>class code</i>	Definizione	<i>class code</i>	Definizione
00 ₁₆	dispositivo realizzato prima della definizione delle classi	01 ₁₆	memoria di massa
02 ₁₆	dispositivo di rete	03 ₁₆	dispositivo di visualizzazione
04 ₁₆	dispositivo multimediale	05 ₁₆	memoria
06 ₁₆	ponte (connessione con altri bus)	07 ₁₆	porta seriale
08 ₁₆	dispositivo di sistema	09 ₁₆	dispositivo di ingresso

<i>class code</i>	Definizione	<i>class code</i>	Definizione
0A ₁₆	<i>docking stations</i>	0B ₁₆	microprocessore
0C ₁₆	bus seriale	0D ₁₆	dispositivo senza fili
0E ₁₆	dispositivo di I/O intelligente	0F ₁₆	comunicazione satellitare
10 ₁₆	dispositivo crittografico	11 ₁₆	acquisizione dati ed elaborazione dei segnali
12 ₁₆ ..FE ₁₆	riservato	FF ₁₆	dispositivo diverso dalle classi esistenti

Tabella 83.107. Interpretazione completa della classe, sottoclasse e interfaccia di programmazione.

<i>class code</i>	<i>subclass</i>	<i>prog if</i>	Descrizione
00 ₁₆	dispositivo realizzato prima della definizione delle classi	01 ₁₆	memoria di massa
00 ₁₆	00 ₁₆	00 ₁₆	dispositivo non meglio precisato, escludendo però il caso di un dispositivo VGA
00 ₁₆	01 ₁₆	00 ₁₆	VGA e compatibili
01 ₁₆	00 ₁₆	00 ₁₆	SCSI
01 ₁₆	01 ₁₆	-- ₁₆	IDE
01 ₁₆	02 ₁₆	00 ₁₆	unità a dischetti
01 ₁₆	03 ₁₆	00 ₁₆	IPI
01 ₁₆	04 ₁₆	00 ₁₆	RAID
01 ₁₆	05 ₁₆	20 ₁₆	PATA, DMA singolo
01 ₁₆	05 ₁₆	30 ₁₆	PATA, DMA concatenato

<i>class code</i>	<i>subclass</i>	<i>prog if</i>	Descrizione
01 ₁₆	06 ₁₆	00 ₁₆	SATA
01 ₁₆	80 ₁₆	00 ₁₆	memoria di massa diversa
02 ₁₆	00 ₁₆	00 ₁₆	Ethernet
02 ₁₆	01 ₁₆	00 ₁₆	Token Ring
02 ₁₆	02 ₁₆	00 ₁₆	FDDI
02 ₁₆	03 ₁₆	00 ₁₆	ATM
02 ₁₆	04 ₁₆	00 ₁₆	ISDN
02 ₁₆	05 ₁₆	00 ₁₆	WorldFip
02 ₁₆	06 ₁₆	--	PICMG 2.14 Multi Computing
02 ₁₆	80 ₁₆	00 ₁₆	interfaccia di rete diversa
03 ₁₆	00 ₁₆	00 ₁₆	VGA
03 ₁₆	00 ₁₆	01 ₁₆	8512
03 ₁₆	01 ₁₆	00 ₁₆	XGA
03 ₁₆	02 ₁₆	00 ₁₆	3D (non VGA)
03 ₁₆	80 ₁₆	00 ₁₆	interfaccia video-grafica diversa
04 ₁₆	00 ₁₆	00 ₁₆	Video multimediale
04 ₁₆	01 ₁₆	00 ₁₆	Audio multimediale
04 ₁₆	02 ₁₆	00 ₁₆	Computer Telephony
04 ₁₆	80 ₁₆	00 ₁₆	interfaccia multimediale diversa
05 ₁₆	00 ₁₆	00 ₁₆	RAM
05 ₁₆	01 ₁₆	00 ₁₆	Flash
05 ₁₆	80 ₁₆	00 ₁₆	interfaccia di memoria diversa
06 ₁₆	00 ₁₆	00 ₁₆	Host Bridge
06 ₁₆	01 ₁₆	00 ₁₆	ISA Bridge
06 ₁₆	02 ₁₆	00 ₁₆	EISA Bridge
06 ₁₆	03 ₁₆	00 ₁₆	MCA Bridge

<i>class code</i>	<i>subclass</i>	<i>prog if</i>	Descrizione
06 ₁₆	00 ₁₆	00 ₁₆	PCI-to-PCI Bridge
06 ₁₆	00 ₁₆	01 ₁₆	PCI-to-PCI Bridge (Subtractive Decode)
06 ₁₆	05 ₁₆	00 ₁₆	PCMCIA Bridge
06 ₁₆	06 ₁₆	00 ₁₆	NuBus Bridge
06 ₁₆	07 ₁₆	00 ₁₆	CardBus Bridge
06 ₁₆	08 ₁₆	-- ₁₆	RACEway Bridge
06 ₁₆	09 ₁₆	40 ₁₆	PCI-to-PCI Bridge (Semi-Transparent, Primary)
06 ₁₆	09 ₁₆	80 ₁₆	PCI-to-PCI Bridge (Semi-Transparent, Secondary)
06 ₁₆	0A ₁₆	00 ₁₆	InfiniBrand-to-PCI Host Bridge
06 ₁₆	80 ₁₆	00 ₁₆	interfaccia di connessione ad altri bus, diversa dai tipi previsti
07 ₁₆	00 ₁₆	00 ₁₆	porta seriale XT
07 ₁₆	00 ₁₆	01 ₁₆	porta seriale 16450
07 ₁₆	00 ₁₆	02 ₁₆	porta seriale 16550
07 ₁₆	00 ₁₆	03 ₁₆	porta seriale 16650
07 ₁₆	00 ₁₆	04 ₁₆	porta seriale 16750
07 ₁₆	00 ₁₆	05 ₁₆	porta seriale 16850
07 ₁₆	00 ₁₆	06 ₁₆	porta seriale 16950
07 ₁₆	01 ₁₆	00 ₁₆	porta parallela
07 ₁₆	01 ₁₆	01 ₁₆	porta parallela bidirezionale
07 ₁₆	01 ₁₆	02 ₁₆	ECP 1.X
07 ₁₆	01 ₁₆	03 ₁₆	IEEE 1284

<i>class code</i>	<i>subclass</i>	<i>prog if</i>	Descrizione
07 ₁₆	01 ₁₆	FE ₁₆	IEEE 1284
07 ₁₆	02 ₁₆	00 ₁₆	unità seriale multipla
07 ₁₆	03 ₁₆	00 ₁₆	modem generico
07 ₁₆	03 ₁₆	01 ₁₆	modem Hayes 16450
07 ₁₆	03 ₁₆	02 ₁₆	modem Hayes 16550
07 ₁₆	03 ₁₆	03 ₁₆	modem Hayes 16650
07 ₁₆	03 ₁₆	04 ₁₆	modem Hayes 16750
07 ₁₆	04 ₁₆	00 ₁₆	IEEE 488.1/2 (GPIB)
07 ₁₆	05 ₁₆	00 ₁₆	Smart Card
07 ₁₆	80 ₁₆	00 ₁₆	interfaccia di comunicazione diversa
08 ₁₆	00 ₁₆	00 ₁₆	8259 PIC
08 ₁₆	00 ₁₆	01 ₁₆	ISA PIC
08 ₁₆	00 ₁₆	02 ₁₆	EISA PIC
08 ₁₆	00 ₁₆	10 ₁₆	I/O APIC
08 ₁₆	00 ₁₆	20 ₁₆	I/O(x) APIC
08 ₁₆	01 ₁₆	00 ₁₆	8237 DMA
08 ₁₆	01 ₁₆	01 ₁₆	ISA DMA
08 ₁₆	01 ₁₆	02 ₁₆	EISA DMA
08 ₁₆	02 ₁₆	00 ₁₆	8254 System Timer
08 ₁₆	02 ₁₆	01 ₁₆	ISA System Timer
08 ₁₆	02 ₁₆	02 ₁₆	EISA System Timer
08 ₁₆	03 ₁₆	00 ₁₆	RTC generico
08 ₁₆	03 ₁₆	01 ₁₆	ISA RTC
08 ₁₆	04 ₁₆	00 ₁₆	unità PCI Hot-Plug generica
08 ₁₆	80 ₁₆	00 ₁₆	dispositivo di sistema diverso
09 ₁₆	00 ₁₆	00 ₁₆	tastiera
09 ₁₆	01 ₁₆	00 ₁₆	<i>digitizer</i>

<i>class code</i>	<i>subclass</i>	<i>prog if</i>	Descrizione
09 ₁₆	02 ₁₆	00 ₁₆	mouse
09 ₁₆	03 ₁₆	00 ₁₆	scanner
09 ₁₆	04 ₁₆	00 ₁₆	gameport
09 ₁₆	04 ₁₆	10 ₁₆	gameport
09 ₁₆	80 ₁₆	00 ₁₆	interfaccia di input diversa
0A ₁₆	00 ₁₆	00 ₁₆	Docking Station
0A ₁₆	80 ₁₆	00 ₁₆	Docking Station diversa
0B ₁₆	00 ₁₆	00 ₁₆	CPU 386
0B ₁₆	01 ₁₆	00 ₁₆	CPU 486
0B ₁₆	02 ₁₆	00 ₁₆	CPU Pentium
0B ₁₆	10 ₁₆	00 ₁₆	CPU Alpha
0B ₁₆	20 ₁₆	00 ₁₆	CPU PowerPC
0B ₁₆	30 ₁₆	00 ₁₆	CPU MIPS
0B ₁₆	40 ₁₆	00 ₁₆	coprocessore
0C ₁₆	00 ₁₆	00 ₁₆	IEEE 1394 (FireWire)
0C ₁₆	00 ₁₆	10 ₁₆	IEEE 1394 (1394 Open-HCI Spec)
0C ₁₆	01 ₁₆	00 ₁₆	ACCESS.bus
0C ₁₆	02 ₁₆	00 ₁₆	SSA
0C ₁₆	03 ₁₆	00 ₁₆	USB (Universal Host)
0C ₁₆	03 ₁₆	10 ₁₆	USB (Open Host)
0C ₁₆	03 ₁₆	20 ₁₆	USB2 Host (Intel Enhanced Host Controller Interface)
0C ₁₆	03 ₁₆	80 ₁₆	USB
0C ₁₆	03 ₁₆	FE ₁₆	USB (Not Host Controller)
0C ₁₆	04 ₁₆	00 ₁₆	Fibre Channel
0C ₁₆	05 ₁₆	00 ₁₆	SMBus

<i>class code</i>	<i>subclass</i>	<i>prog if</i>	Descrizione
0C ₁₆	06 ₁₆	00 ₁₆	InfiniBand
0C ₁₆	07 ₁₆	00 ₁₆	IPMI SMIC
0C ₁₆	07 ₁₆	01 ₁₆	IPMI Kybd
0C ₁₆	07 ₁₆	02 ₁₆	IPMI Block Transfer
0C ₁₆	08 ₁₆	00 ₁₆	SERCOS (IEC 61491)
0C ₁₆	09 ₁₆	00 ₁₆	CANbus
0D ₁₆	00 ₁₆	00 ₁₆	iRDA
0D ₁₆	01 ₁₆	00 ₁₆	Consumer IR
0D ₁₆	10 ₁₆	00 ₁₆	RF Controller
0D ₁₆	11 ₁₆	00 ₁₆	Bluetooth
0D ₁₆	12 ₁₆	00 ₁₆	Broadband
0D ₁₆	20 ₁₆	00 ₁₆	Ethernet WiFi (802.11a)
0D ₁₆	21 ₁₆	00 ₁₆	Ethernet WiFi (802.11b)
0D ₁₆	80 ₁₆	00 ₁₆	interfaccia di rete senza fili, diversa da quelle previste
0E ₁₆	00 ₁₆	--	I20 Architecture
0E ₁₆	00 ₁₆	00 ₁₆	Message FIFO
0F ₁₆	01 ₁₆	00 ₁₆	TV
0F ₁₆	02 ₁₆	00 ₁₆	Audio
0F ₁₆	03 ₁₆	00 ₁₆	Voice
0F ₁₆	04 ₁₆	00 ₁₆	Data
10 ₁₆	00 ₁₆	00 ₁₆	Network and Computing Encryption/Decryption
10 ₁₆	10 ₁₆	00 ₁₆	Entertainment Encryption/Decryption
10 ₁₆	80 ₁₆	00 ₁₆	Other Encryption/Decryption
11 ₁₆	00 ₁₆	00 ₁₆	DPIO Modules
11 ₁₆	01 ₁₆	00 ₁₆	Performance Counters

<i>class code</i>	<i>subclass</i>	<i>prog if</i>	Descrizione
11 ₁₆	10 ₁₆	00 ₁₆	Communications Synchronization Plus Time and Frequency Test/Measurement
11 ₁₆	20 ₁₆	00 ₁₆	Management Card
11 ₁₆	80 ₁₆	00 ₁₆	interfaccia di acquisizione dati o di elaborazione dei segnali, diversa da quelle previste

83.10.3 Raccolta delle informazioni

Per raccogliere le informazioni sui dispositivi connessi a un bus PCI, occorre predisporre un selettore. Per esempio, utilizzando una variabile strutturata di tipo `'pci_address_t'` si potrebbe richiedere di accedere al bus *b*, al dispositivo *s* (*slot*), alla funzione *f* e al registro *r*:



```

int          b;
int          s;
int          f;
int          r;
pci_address_t pci_addr;
uint32_t     reg;
...
pci_addr.selector = 0;
pci_addr.enable   = 1;
pci_addr.bus      = b;
pci_addr.slot     = s;
pci_addr.function = f;
pci_addr.reg      = r;
out_32 (0x0CF8, pci_addr.selector);
reg = in_32 (0x0CFC);

```

Nell'esempio, le funzioni *out_32()* e *in_32()* utilizzano in pratica le istruzioni 'OUTL' e 'INL' del linguaggio assembler (si vedano eventualmente i listati [94.6](#), [94.6.5](#) e [94.6.2](#)). Alla fine, la variabile *reg* raccoglie il contenuto del registro selezionato.

Per scandire un bus PCI è possibile procedere provando tutte le combinazioni di bus, alloggiamento e funzione, verificando che il primo registro sia diverso da $0xFFFFFFFF_{16}$. Se è così si può raccogliere il contenuto della tabella corrispondente. Nell'esempio seguente si scandiscono tutti i bus PCI, ignorando i dispositivi di classe 06_{16} , raccogliendo alcuni dati dei dispositivi validi in un array con elementi di tipo 'struct pci'. Si esclude che si possano incontrare dispositivi con più funzioni; inoltre si ritiene di incontrare soltanto dispositivi che contengono nel campo *BAR0* una porta di I/O.

```

struct {
    unsigned char    bus;
    unsigned char    slot;

```



```
    unsigned short int vendor_id;
    unsigned short int device_id;
    unsigned char    class_code;
    unsigned char    subclass;
    unsigned char    prog_if;
    uintptr_t        base_io;
    unsigned char    irq;
} pci[8];
pci_header_type_00_t pci;
pci_address_t        pci_addr;
//
int t;                // PCI table index.
int b;                // PCI bus index.
int s;                // PCI slot index.
int r;                // PCI header register index.
//
// Reset the PCI table.
//
for (t = 0; t < 8; t++)
{
    pci_table[t].bus          = 0;
    pci_table[t].slot        = 0;
    pci_table[t].vendor_id   = 0;
    pci_table[t].device_id   = 0;
    pci_table[t].class_code  = 0;
    pci_table[t].subclass    = 0;
    pci_table[t].prog_if     = 0;
    pci_table[t].base_io     = 0;
    pci_table[t].irq         = 0;
}
//
// Scan PCI buses and slots.
//
t = 0;
```

```
//
for (b = 0; b < 256 && t < 8; b++)
{
    for (s = 0; s < 32 && t < 8; s++)
    {
        pci_addr.selector    = 0;
        pci_addr.enable     = 1;
        pci_addr.bus        = b;
        pci_addr.slot       = s;
        //
        pci_addr.reg        = 0;
        out_32 (0x0CF8, pci_addr.selector);
        pci.r[0] = in_32 (0x0CFC);
        //
        if (pci.r[0] == 0xFFFFFFFF)
        {
            //
            // There is no such bus:slot combination!
            //
            continue;
        }
        else
        {
            for (r = 1; r < 16; r++)
            {
                pci_addr.reg        = r;
                out_32 (0x0CF8, pci_addr.selector);
                pci.r[r] = in_32 (0x0CFC);
            }
        }
        //
        // We consider only PCI header type 0x00!
        //
        if (pci.header_type != 0)
```

```
        {
            continue;
        }
    //
    // We do not consider PCI bridge devices!
    //
    if (pci.class_code == 0x06)
    {
        continue;
    }
    //
    // Save the device inside the PCI table.
    //
    pci_table[t].bus          = b;
    pci_table[t].slot        = s;
    pci_table[t].vendor_id   = pci.vendor_id;
    pci_table[t].device_id   = pci.device_id;
    pci_table[t].class_code  = pci.class_code;
    pci_table[t].subclass    = pci.subclass;
    pci_table[t].prog_if     = pci.prog_if;
    pci_table[t].base_io     = pci.bar0 & 0xFFFFFFFFFC;
    pci_table[t].irq         = pci.interrupt_line;
    //
    // Next PCI table row.
    //
    t++;
    }
}
```

Come si può osservare, la presunta porta di I/O viene filtrata con una maschera, in modo da azzerare i due bit meno significativi:

```
pci_table[t].base_io = pci.bar0 & 0xFFFFFFFFFC;
```

Al termine della scansione, la combinazione dei codici identificativi del produttore e del dispositivo, permettono di sapere di che cosa si tratta, disponendo naturalmente di un elenco appropriato. Per esempio, il produttore $10EC_{16}$ corrisponde a Realtek Semiconductor, mentre il dispositivo 8029_{16} (relativo a tale produttore) corrisponde a un'interfaccia di rete RT8029, compatibile con la vecchia NE2000.

83.11 NE2000

«

Le interfacce di rete NE2000 hanno delle limitazioni significative e sono complesse da programmare. Tuttavia, questo tipo di dispositivo è quello più facilmente disponibile negli emulatori, per esempio è presente sia in Bochs, sia in Qemu, pertanto diventa una scelta obbligata, almeno inizialmente.

Le annotazioni fatte qui, a proposito delle interfacce di rete NE2000, sono insufficienti per una gestione completa di tali unità. Eventualmente si possono consultare due schede tecniche, citate anche alla fine del capitolo: *DP8390D/NS32490D NIC network interface controller* e *Writing drivers for DP8390 NIC family of ethernet controllers*.

83.11.1 Memoria interna

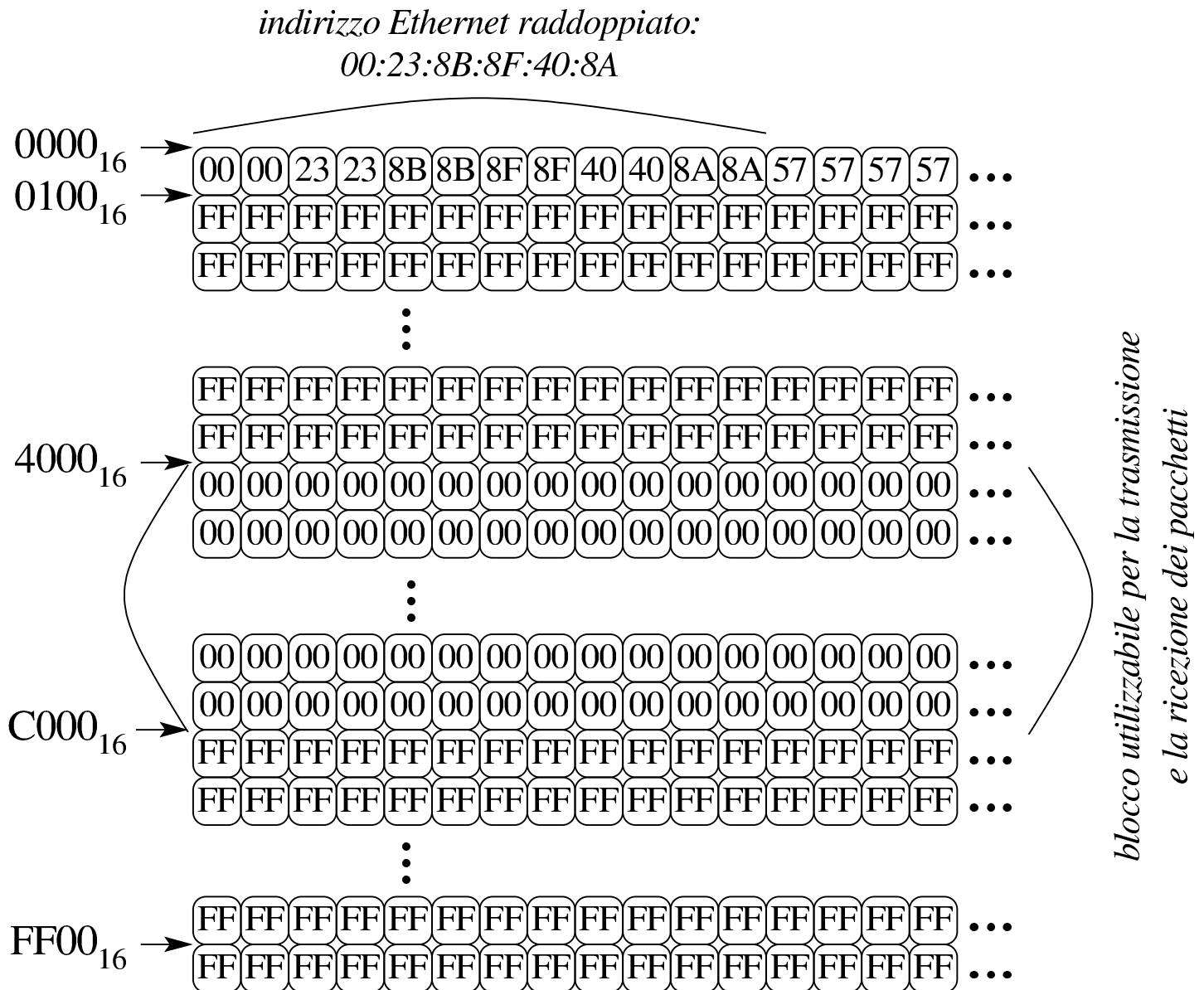
«

Un aspetto importante della programmazione dell'interfaccia di rete NE2000 riguarda la memoria interna, complessivamente di 64 Kibyte, entro la quale va individuata una porzione per i pacchetti da trasmettere (in questo contesto sono più precisamente trame) e un'altra per la coda di ricezione. Per accedere a questa memoria, sia in scrittura, sia in lettura, occorrono dei comandi opportuni, per poi eseguire l'operazione attraverso una porta di I/O.

La memoria interna è organizzata a blocchi da 256 byte (100_{16}), perché alcuni registri usati come puntatori possono farvi riferimento, disponendo solo di 8 bit.

La porzione iniziale di questa memoria contiene delle informazioni importanti sull'interfaccia; in particolare è annotato lì l'indirizzo Ethernet attribuito dal costruttore. Va osservato che la porzione utile per la collocazione dei pacchetti da trasmettere o da ricevere va dall'indirizzo 4000_{16} a $BFFF_{16}$ (estremi inclusi); il resto deve essere lasciato alla gestione interna dell'interfaccia.

Figura 83.111. Organizzazione della memoria tampone interna di un'interfaccia di rete NE2000.

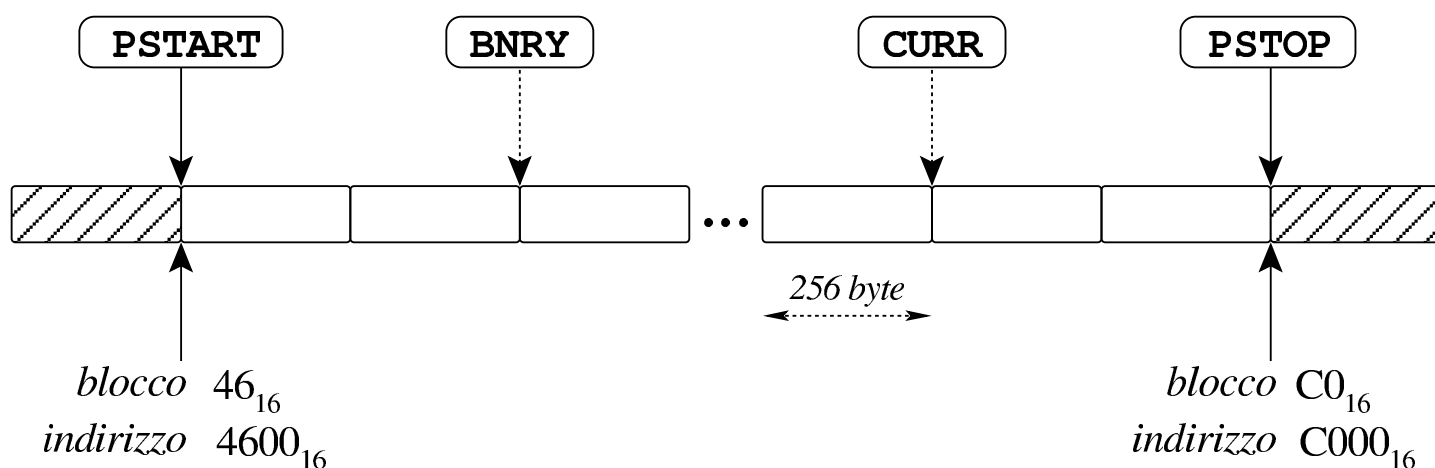


Per un'interfaccia di rete NE2000, il contenuto di un pacchetto, esclusa l'intestazione Ethernet, può andare da un minimo di 46 byte a un massimo di 1500 byte. Data l'organizzazione della memoria interna dell'interfaccia, un pacchetto utilizza da uno a sei blocchi da 256 byte (se un pacchetto è molto breve, utilizza ugualmente un blocco intero di memoria).

83.11.2 Coda di ricezione

Quando l'interfaccia di rete riceve un pacchetto, lo colloca in una porzione della propria memoria tampone, organizzata in forma di coda, a blocchi di 256 byte. Questa porzione di memoria viene chiamato «anello», perché una volta raggiunto l'ultimo blocco, si riprende dal primo.

Figura 83.112. Esempio di coda per la ricezione dei pacchetti, dal byte 4600_{16} al $BFFF_{16}$, ovvero dal blocco 46_{16} al BF_{16} .



La zona di memoria da usare per la coda è delimitata dal valore di due registri, **PSTART** (*page start*) e **PSTOP** (*page stop*). Seguendo l'esempio che si vede nella figura, **PSTART** contiene il valore 46_{16} , mentre **PSTOP**₁₆ ha il valore $C0_{16}$.

Mano a mano che la scrittura nella coda procede, viene incrementato un indice rappresentato dal registro **CURR** (*current page*), il quale rappresenta il prossimo blocco di memoria da utilizzare per la scrittura. Nell'esempio, il registro **CURR** contiene il valore BE_{16} , corrispondente al penultimo blocco.

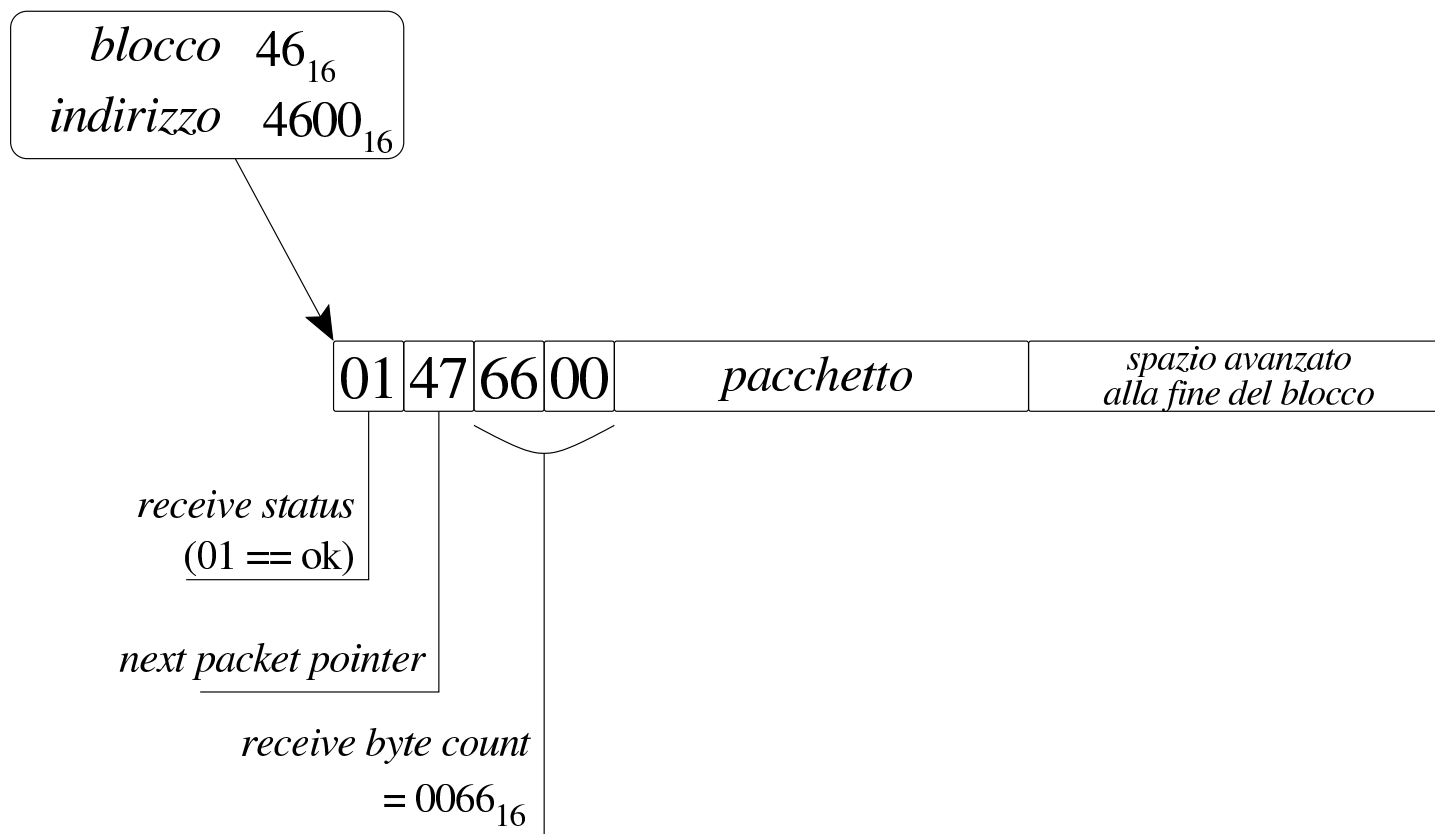
Per la lettura dei blocchi si usa l'indice rappresentato dal registro **BNRY** (*boundary pointer*), il quale si riferisce al prossimo blocco

ancora da leggere. La lettura dei blocchi si deve arrestare quando l'indice ***BNRY*** raggiunge l'indice ***CURR***; per converso, la ricezione dei pacchetti si deve arrestare quando l'indice ***CURR*** raggiunge l'indice ***BNRY***, anche se ciò comporta la perdita di pacchetti.

Quando la ricezione di un pacchetto fa sì che l'indice ***CURR*** raggiunga l'indice ***BNRY***, senza avere completato la ricezione, si ottiene uno straripamento, ma la porzione di pacchetto depositata nella coda non viene rimossa.

All'inizio di ogni pacchetto ricevuto appaiono 4 byte di intestazione, con le informazioni che si possono vedere nella figura successiva. A seconda di come avviene la lettura, l'ordine dei dati può variare: nella figura si ipotizza un accesso a byte singoli.

Figura 83.113. Esempio di blocco collocato all'indirizzo 4600_{16} , contenente un'intestazione e un pacchetto abbastanza corto da non superare il blocco stesso. Si osservi che la dimensione riportata nel terzo e quarto byte dell'intestazione, include i quattro byte dell'intestazione stessa.



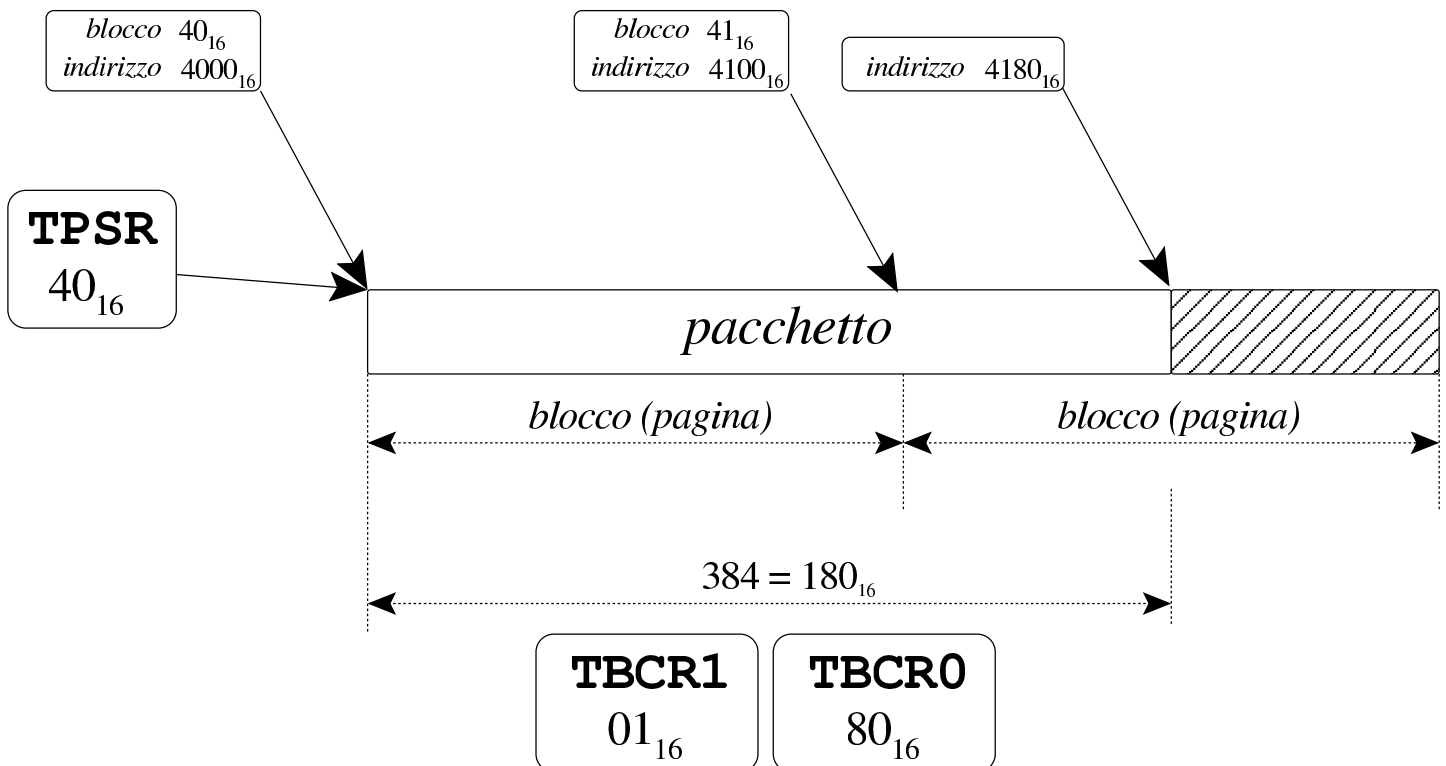
La figura ipotizza che nel blocco di memoria 46_{16} , pari all'indirizzo 4600_{16} , sia stato annotato un pacchetto ricevuto, i cui primi quattro byte indicano una ricezione corretta e una dimensione di 0066_{16} byte. Se successivamente a questo pacchetto ne è stato ricevuto un altro, questo lo si trova a partire dal blocco 47_{16} , come annotato nel secondo byte dell'intestazione del primo.

83.11.3 Tampone di trasmissione

« Sapendo che un pacchetto può occupare al massimo sei blocchi di memoria, per la trasmissione è sufficiente riservare un'area di sei blocchi, per esempio da 4000_{16} a $45FF_{16}$ (estremi inclusi).

Una volta collocato il pacchetto da trasmettere nella memoria interna dell'interfaccia, occorre indicare il blocco iniziale in cui si trova il pacchetto e la sua dimensione in byte, attraverso i registri **TPSR** (*transmit page start*), **TBCR0** e **TBCR1** (*transit byte count*), come si vede nella figura successiva.

Figura 83.114. Esempio di blocco da trasmettere, collocato all'indirizzo 4000_{16} , lungo 384 byte (un blocco e mezzo), con i valori da assegnare ai registri responsabili per l'invio.

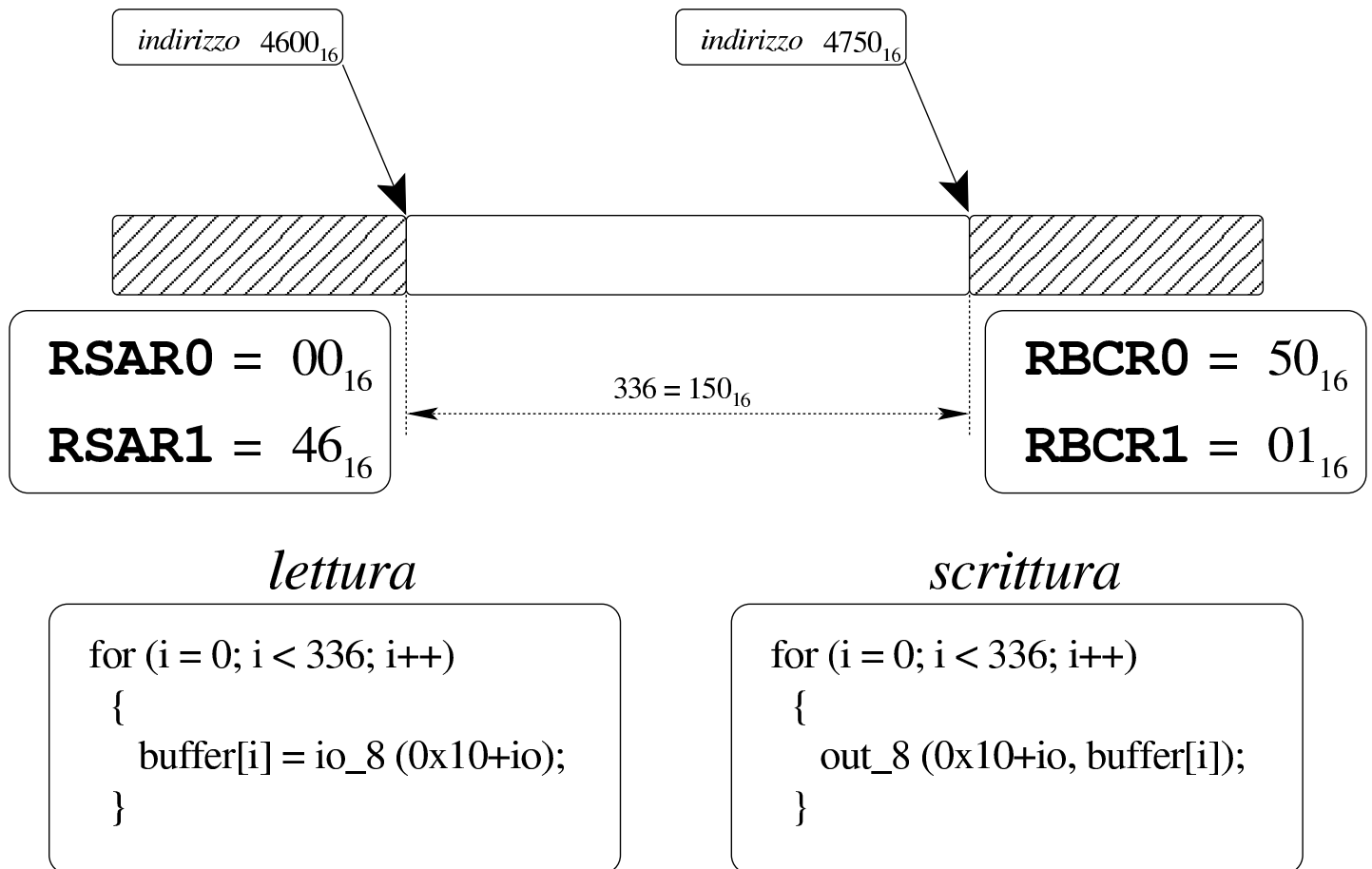


Si osservi che il pacchetto da trasmettere contiene solo ciò che serve per una trama Ethernet; pertanto, quei quattro byte di intestazione che si trovano in ricezione, non ci sono affatto in trasmissione.

83.11.4 Trasferimento dati tra la memoria interna e quella dell'elaboratore

I pacchetti ricevuti e quelli da trasmettere, si trovano necessariamente nella memoria interna dell'interfaccia. Per trasferire i dati tra la memoria interna a quella dell'elaboratore, occorre comunicare attraverso la porta di comunicazione 10_{16} , più l'indirizzo relativo dell'interfaccia, ma prima va definita la posizione iniziale nella memoria interna, attraverso i registri ***RSAR0*** e ***RSAR1*** (*remote start address*), inoltre va specificata la quantità di byte da trasferire, con i registri ***RBCR0*** e ***RBCR1*** (*remote byte count*).

Figura 83.115. Lettura o scrittura di un'area di memoria interna. La variabile *io* dell'esempio rappresenta l'indirizzo di I/O dell'interfaccia, a cui poi si somma l'indirizzo relativo della porta di comunicazione (10_{16}).



83.11.5 Registri e porte

«

Per la gestione dell'interfaccia di rete NE2000 è necessario accedere a dei registri, leggendo o scrivendo dei dati. Questi registri si raggiungono attraverso delle porte di I/O, di cui si conosce lo scostamento rispetto a un indirizzo iniziale. Per esempio, se l'interfaccia è configurata complessivamente per operare a partire dalla porta 0300_{16} , dovendo intervenire con la porta «dati», già vista in precedenza, che si trova nell'indirizzo relativo 10_{16} , in pratica occorre comunicare con la porta 0310_{16} . Quando si utilizza un'interfaccia

connessa a un bus PCI, si ottiene l'indirizzo della porta iniziale dal campo **BAR0**, azzerando i due bit meno significativi (sezione 83.10).

I registri sono raggruppati in tre pagine, numerate da zero a due, ma in ogni pagina si distingue tra registri in lettura o in scrittura. Nei casi più semplici, lo stesso registro è accessibile in lettura e scrittura nella stessa pagina, come nel caso del registro **CURR**, nella pagina uno; in altre situazioni le cose si complicano, come nel caso dei registri **PSTART** e **PSTOP** a cui si accede in scrittura nella pagina zero, oppure in lettura nella pagina due.

Dal momento che con una stessa porta di comunicazione si possono individuare registri differenti, prima occorre selezionare una pagina, attraverso un comando che si impartisce con il registro **CR** (*command register*), il quale ha la particolarità di essere accessibile in tutte le pagine.

Tabella 83.116. Registri NE2000.

Scostamento	pagina 0 lettura	pagina 0 scrittura	pagina 1 lettura	pagina 1 scrittura	pagina 2 lettura	pagina 2 scrittura
00 ₁₆	CR	CR	CR	CR	CR	CR
01 ₁₆	CLDA0	PSTART	PAR0	PAR0	PSTART	CLDA0
02 ₁₆	CLDA1	PSTOP	PAR1	PAR1	PSTOP	CLDA1
03 ₁₆	BNRY	BNRY	PAR2	PAR2	--	--
04 ₁₆	TSR	TPSR	PAR3	PAR3	TPSR	--
05 ₁₆	NCR	TBCR0	PAR4	PAR4	--	--
06 ₁₆	FIFO	TBCR1	PAR5	PAR5	--	--
07 ₁₆	ISR	ISR	CURR	CURR	--	--
08 ₁₆	CRDA0	RSAR0	MAR0	MAR0	--	--
09 ₁₆	CRDA1	RSAR1	MAR1	MAR1	--	--

Scostamento	pagina 0 lettura	pagina 0 scrittura	pagina 1 lettura	pagina 1 scrittura	pagina 2 lettura	pagina 2 scrittura
0A ₁₆	--	RBCR0	MAR2	MAR2	--	--
0B ₁₆	--	RBCR1	MAR3	MAR3	--	--
0C ₁₆	RSR	RCR	MAR4	MAR4	RCR	--
0D ₁₆	CNTR0	TCR	MAR5	MAR5	TCR	--
0E ₁₆	CNTR1	DCR	MAR6	MAR6	DCR	--
0F ₁₆	CNTR2	IMR	MAR7	MAR7	IMR	--

Tabella 83.117. Altre porte di comunicazione nelle interfacce di rete NE2000.

Scostamento	Descrizione
10 ₁₆ -17 ₁₆	Accesso alla memoria interna (<i>remote DMA</i>).
18 ₁₆ -1F ₁₆	Azzeramento.

Figura 83.118. Sintesi dell'utilizzo del registro *CR*, *command register*.

CR – command register

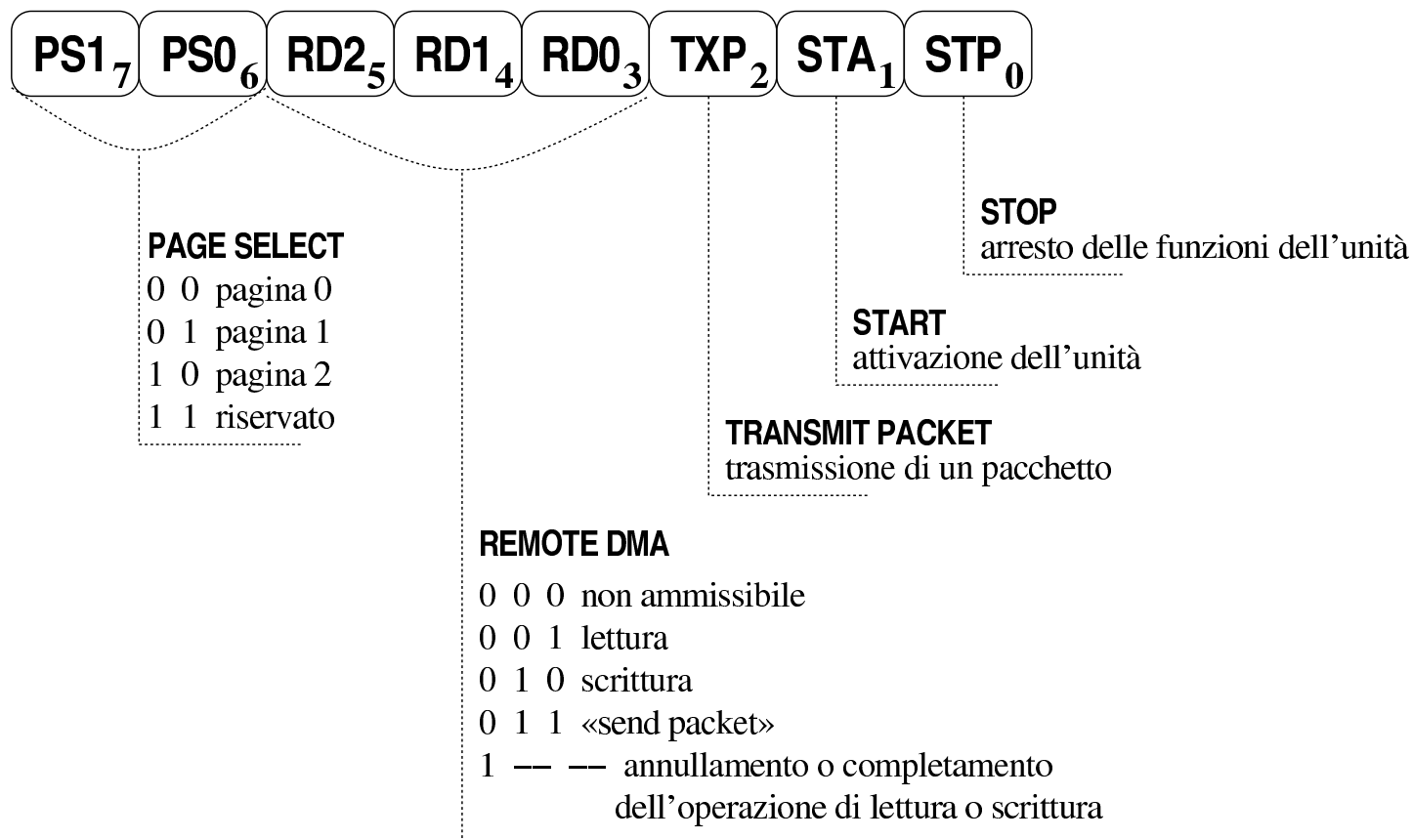


Figura 83.119. Sintesi dell'utilizzo del registro *ISR*, *interrupt status register*.

ISR – interrupt status register

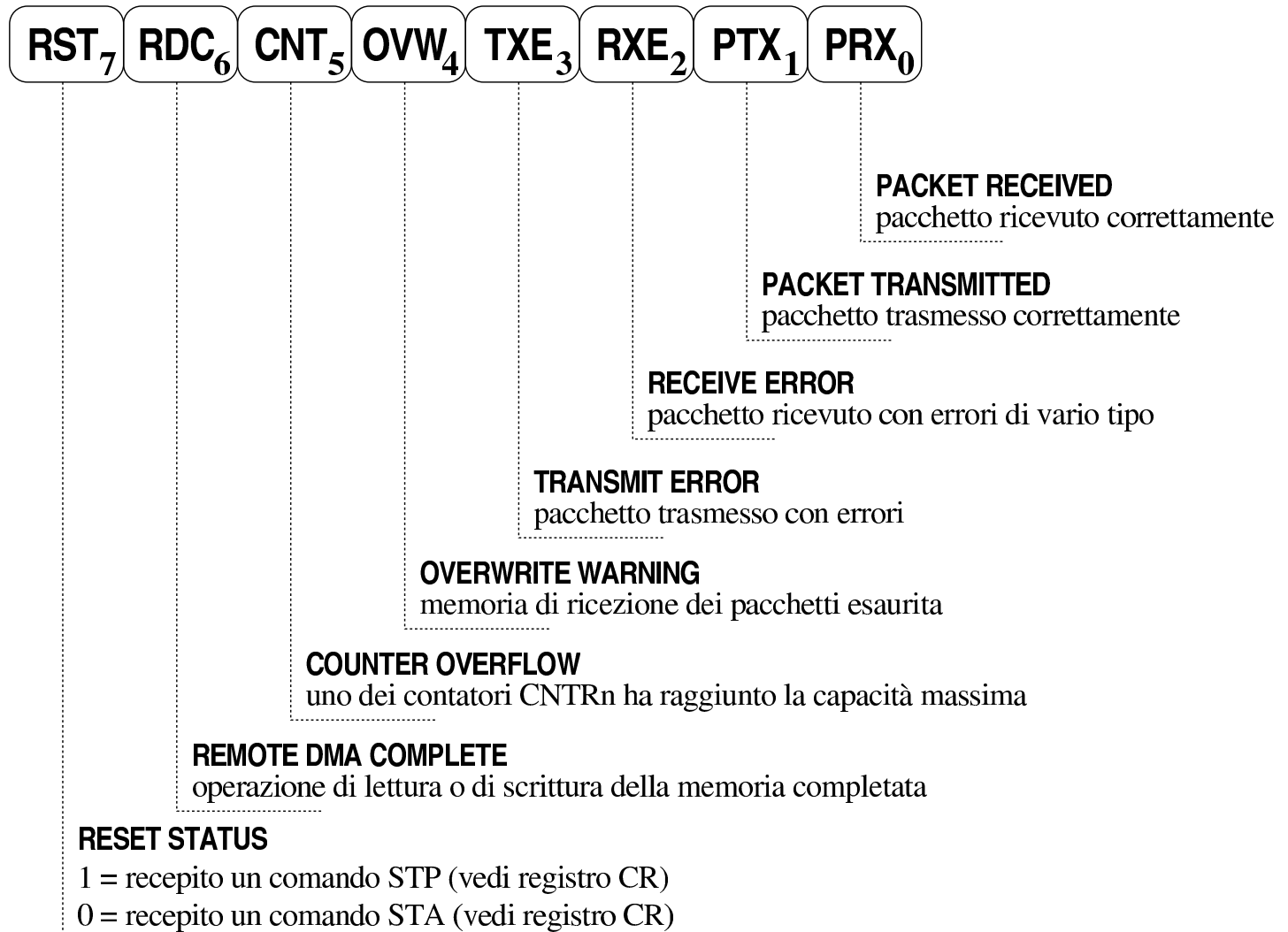


Figura 83.120. Sintesi dell'utilizzo del registro *DCR*, *data configuration register*.

DCR – data configuration register

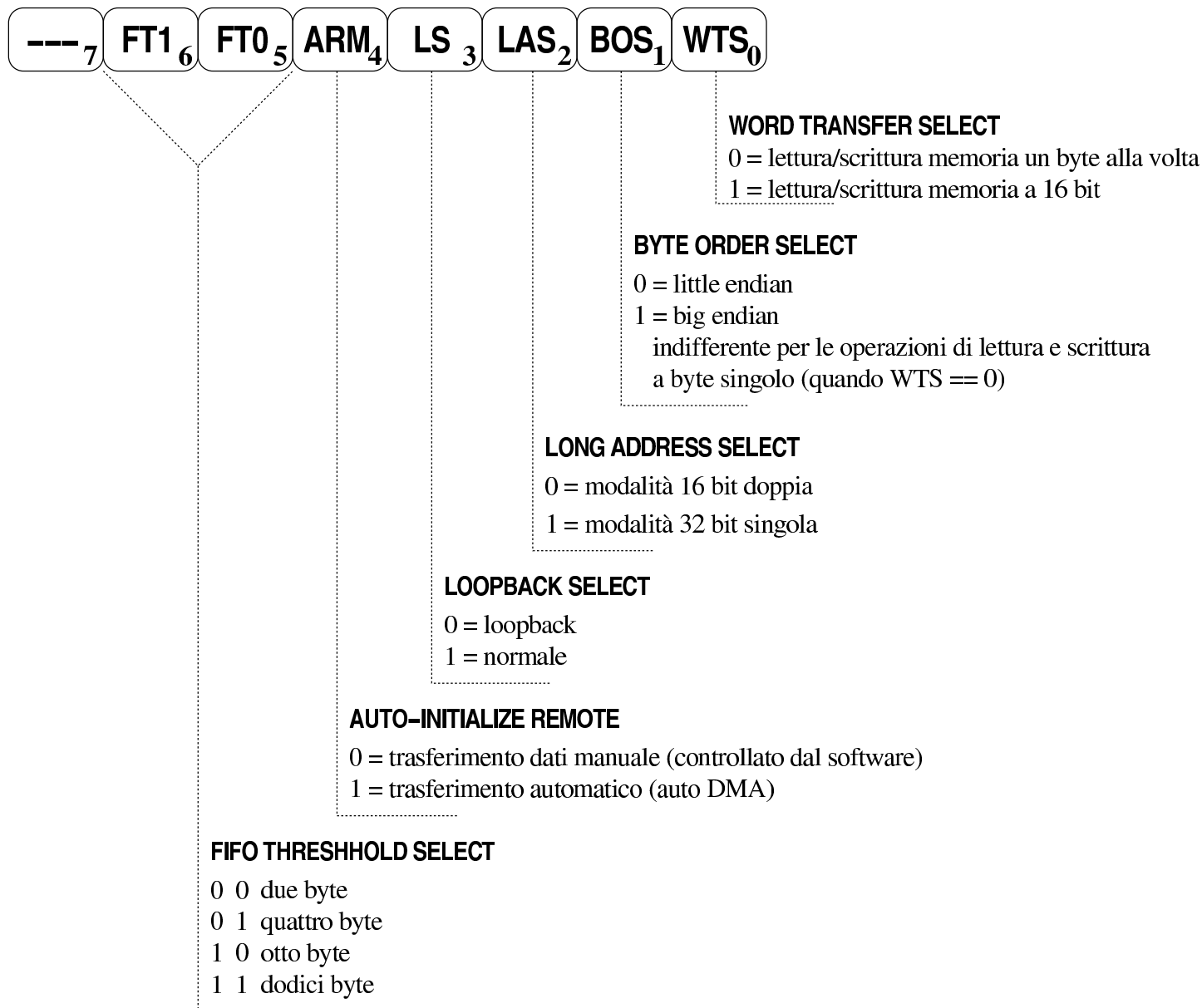


Figura 83.121. Sintesi dell'utilizzo del registro *TCR*, *transmit configuration register*.

TCR – transmit configuration register

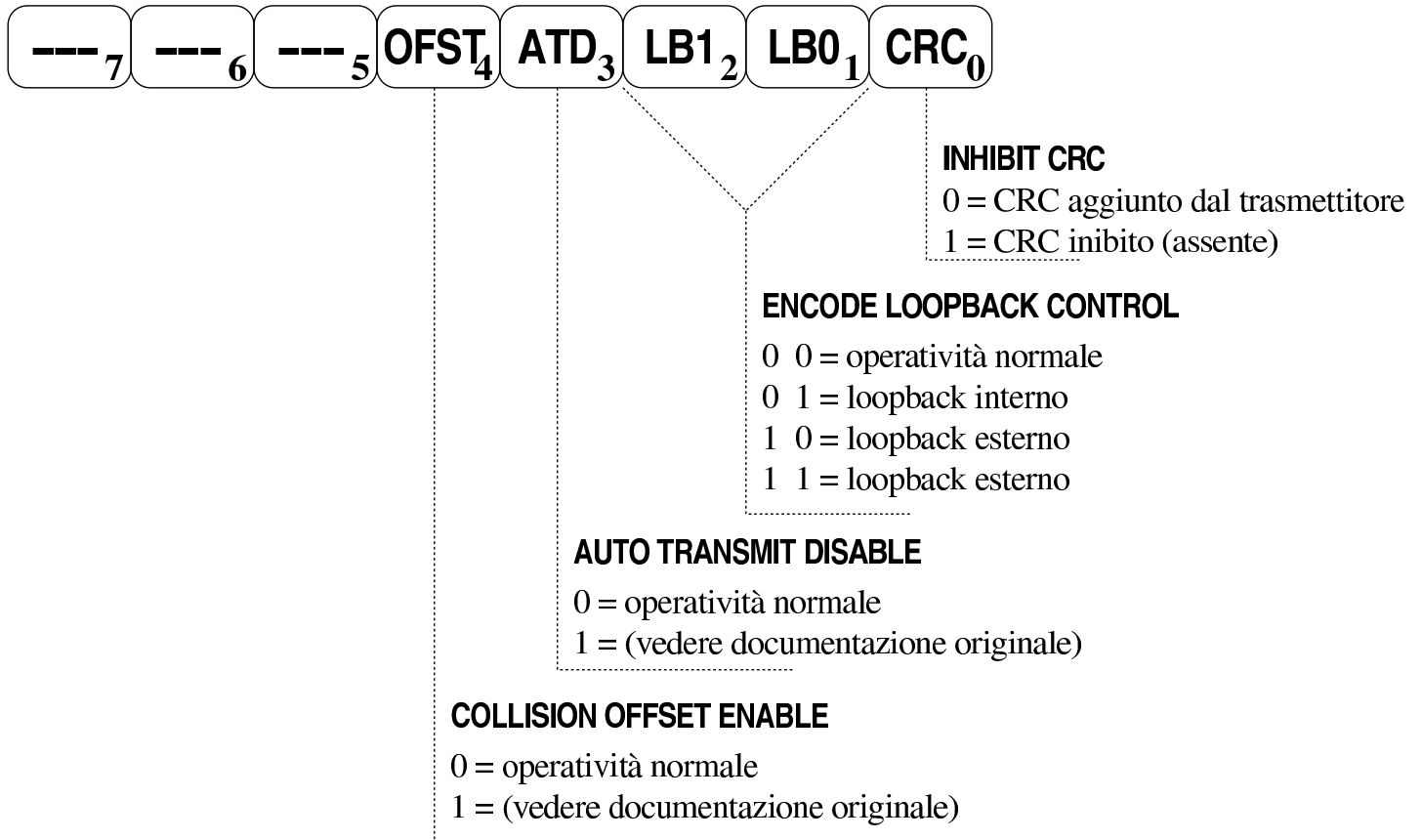


Figura 83.122. Sintesi dell'utilizzo del registro *TSR*, *transmit status register*.

TSR – transmit status register

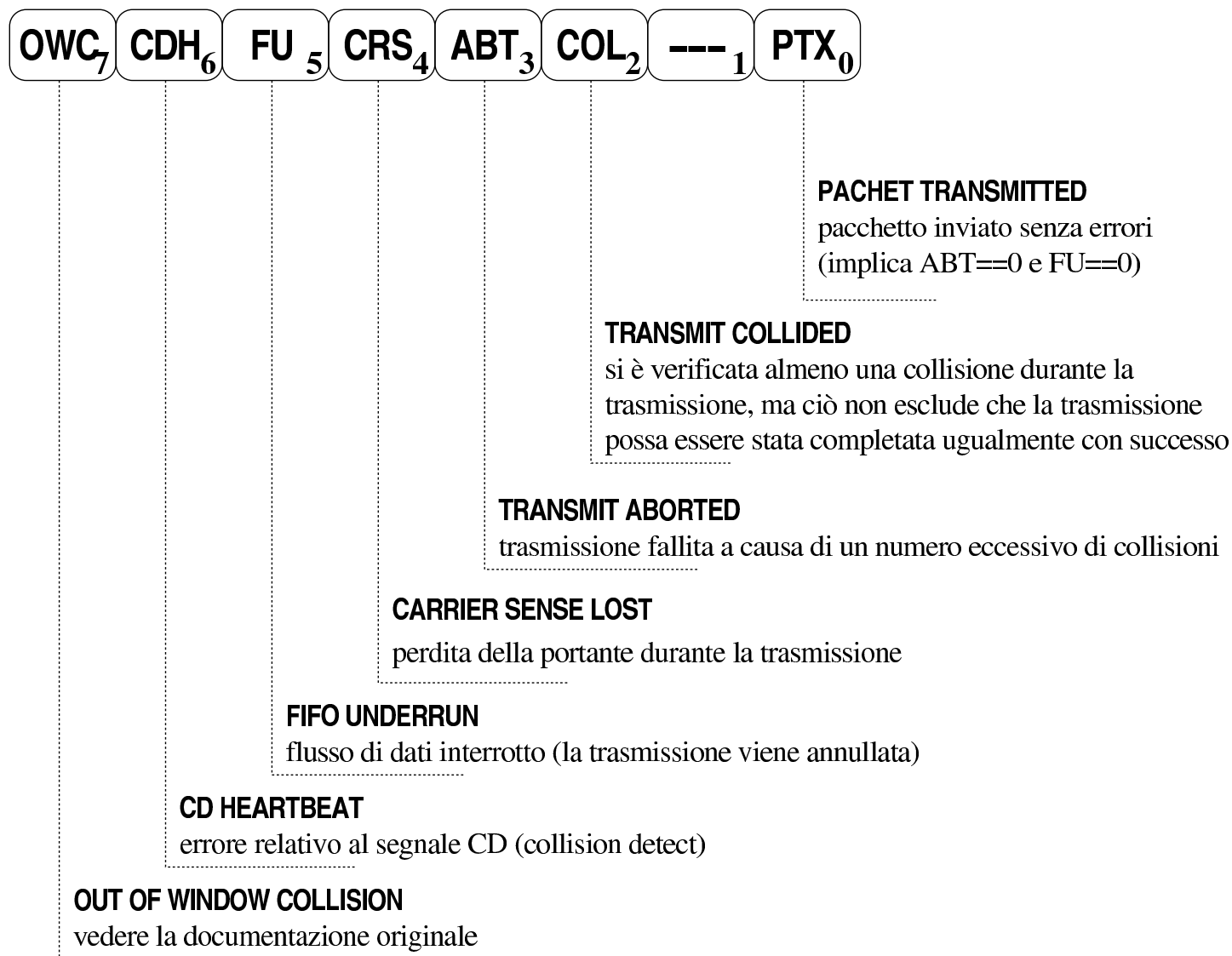


Figura 83.123. Sintesi dell'utilizzo del registro *RCR*, *receive configuration register*.

RCR – receive configuration register

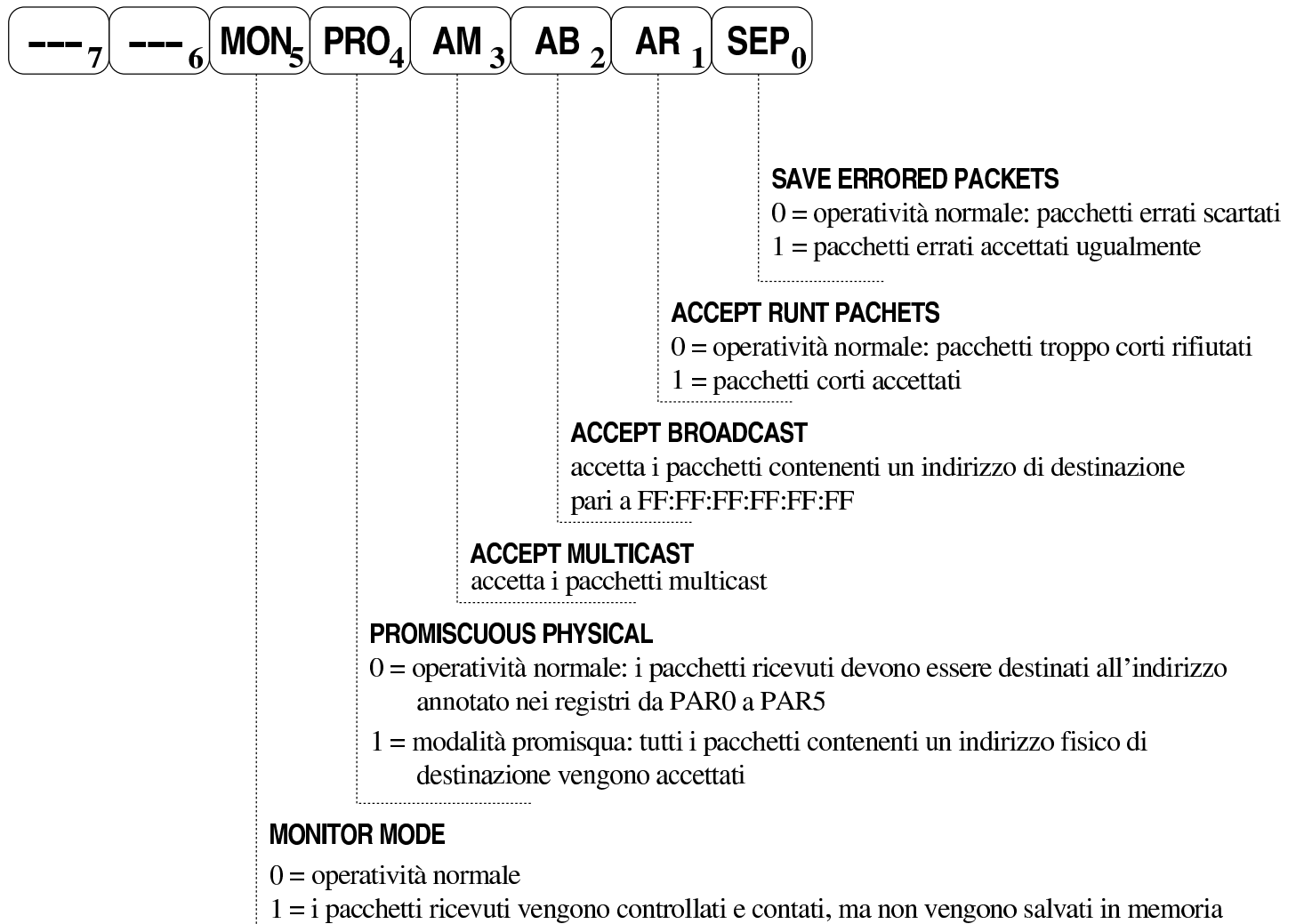
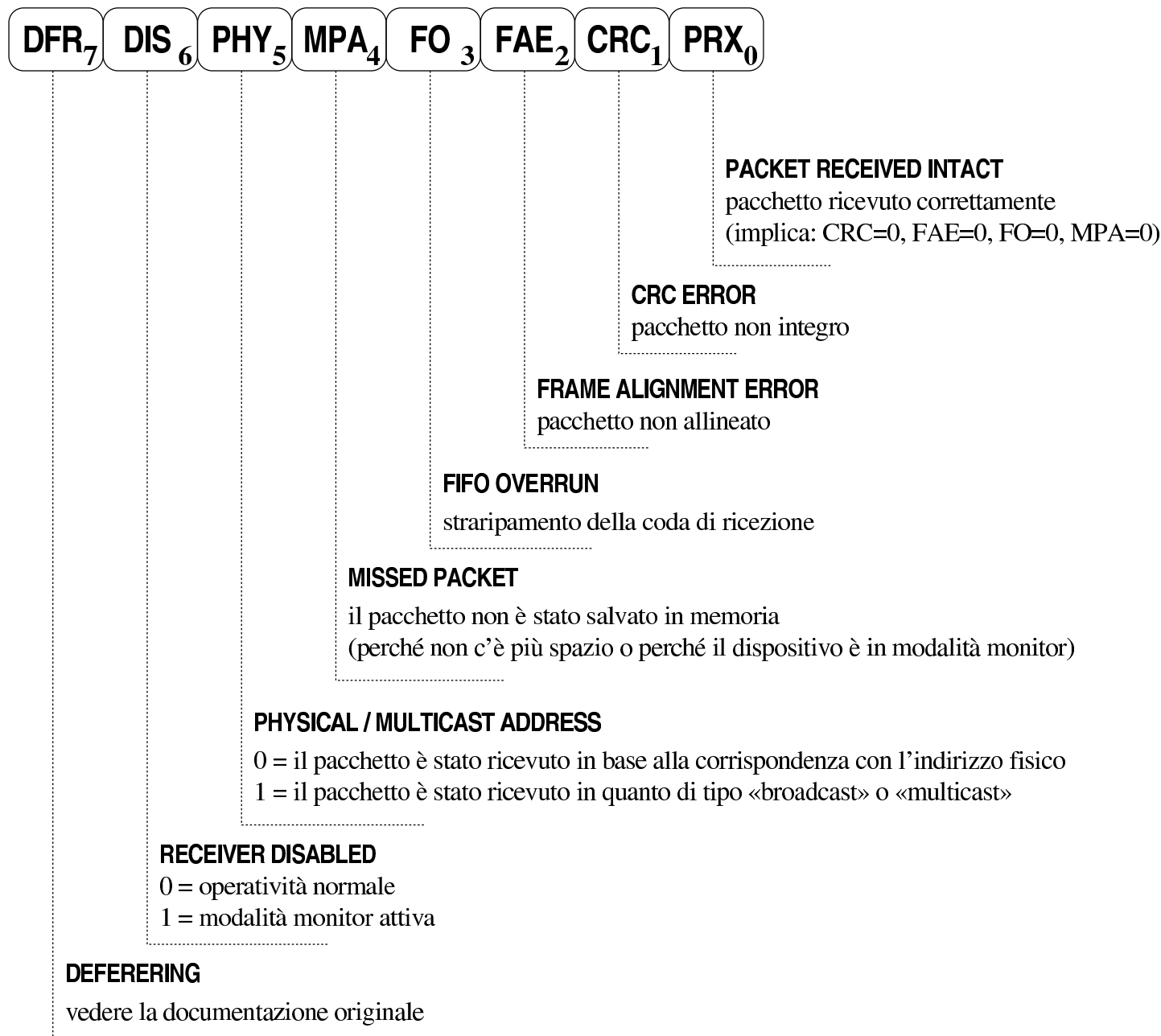


Figura 83.124. Sintesi dell'utilizzo del registro *RSR*, *receive status register*.

RSR – receive status register



83.11.6 Procedura di riconoscimento

«

Nel listato successivo si annota una procedura utile per riconoscere un'interfaccia NE2000, partendo dalla porta di I/O di partenza. Il procedimento è stato determinato leggendo il codice del kernel Linux, precisamente nel file '*linux/drivers/net/ne2k-pci.c*'. La variabile *io* rappresenta la porta di I/O iniziale, per accedere all'interfaccia; le funzioni *in_8()* e *out_8()* servono rispettivamente per leggere un byte da una porta o per scrivercelo.

```

int status;
int reg_00;
int reg_0d;
//
// Legge il registro CR (0x00), il quale non può restituire
// un valore pari a 0xFF.
//
reg_00 = in_8 (io + 0x00);
if (reg_00 == 0xFF)
{
    return (-1);
}
//
// Tenta di selezionare la seconda pagina di registri,
// attraverso la modifica del registro CR (0x00).
//
// -----
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|
// | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0x61
// '-----'
//   \___/   \_____/   |
//   |       |       |   STOP
//   |       |       |   Abort/complete

```

```

//      |          remote DMA
//      Register
//      page 1
//
out_8 ((io + 0x00), 0x61);
//
// Legge e poi modifica il contenuto del registro MAR4 (0x0D
// nella seconda pagina).
//
reg_0d = in_8 (io + 0x0D);
out_8 (io + 0x0D, 0xFF);
//
// Tenta nuovamente di selezionare la prima pagina di
// registri.
//
// .------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|
// | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
// |-----'
//      \___/      \_____/      |
//      |          |          STOP
//      |          Abort/complete
//      |          remote DMA
//      Register
//      page 0
//
out_8 ((io + 0x00), 0x21);
//
// Legge il contatore CNTR0 (0x0D).
//
in_8 (io + 0x0D);
//
// La lettura del contatore dovrebbe averlo azzerato: lo si

```

```
// rilegge per verificare questo fatto.
//
status = in_8 (io + 0x0D);
if (status)
{
    //
    // Dal momento che il contatore non si è azzerato, non
    // può trattarsi di un'interfaccia NE2000, e molto
    // probabilmente lo scambio da una pagina all'altra di
    // registri non ha avuto effetto; per cui, viene
    // ripristinato ciò che era contenuto
    // nella posizione 0x00 e 0x0D.
    //
    out_8 (io, reg_00);
    out_8 ((io + 0x0D), reg_0d);
    return (-1);
}
//
// Se non si sono verificati casi di errore, è plausibile
// che si tratti di un'interfaccia NE2000.
//
return (0);
```

83.11.7 Procedura di inizializzazione

«

Una volta chiarito che alla porta *io* risponde un'interfaccia NE2000, si può procedere con la sua inizializzazione. Durante questa fase è importante determinare l'indirizzo fisico dell'interfaccia, il quale va poi trascritto nei registri da **PAR0** a **PAR5**; inoltre si stabilisce la zona della memoria interna utilizzata per i pacchetti da trasmettere e per la coda di ricezione: viene usato lo stesso schema già apparso nella figura 83.111.


```

int      status;
int      i;
uint8_t sa_prom[12];
uint8_t par[6];
//
// Viene azzerata l'interfaccia, scrivendo nella porta 0x1F.
//
status = in_8 (io + 0x1F);
out_8 ((io + 0x1F), 0xFF);
out_8 ((io + 0x1F), status);
//
// Dopo l'azzeramento occorre attendere che il registro
// ISR (0x07) segnali il recepimento di questo stato.
//
// Interrupt status register (ISR)
// -----
// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-----|
// | 1 | ? | ? | ? | ? | ? | ? | ? | 0x80
// '-----'
// |
// Reset status
//
while (1)
{
    if (in_8 (io + 0x07) & 0x80) // ISR
    {
        //
        // Il registro ISR segnala l'azzeramento avvenuto e
        // si esce dal ciclo di attesa.
        //
        break;
    }
}

```

```

//
// Azzerare tutti gli indicatori del registro ISR (0x07).
//
out_8 ((io + 0x07), 0xFF); // ISR
//
// Si procede con la lettura dell'indirizzo fisico,
// contenuto nella «SA-PROM» (station address PROM).
// Per prima cosa si imposta il registro CR (0x00),
// richiedendo la conclusione di ogni attività di
// trasferimento dati (in questo caso
// sarebbe superfluo, ma non può far male).
//
// Command register (CR)
// .-----.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|
// | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
// '-----'
//  \_____/ \_____/ |
//  | | | STOP
//  | | |
//  | Abort/complete
//  | remote DMA
//  |
// Register
// page 0
//
out_8 ((io + 0x00), 0x21); // CR
//
// Come già avvenuto in precedenza, si attende che il
// registro ISR (0x07) confermi l'avvenuto azzeramento.
//
// Interrupt status register (ISR)
// .-----.

```

```

// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-----|
// | 1 | ? | ? | ? | ? | ? | ? | ? | 0x80
// `-----'
// |
// Reset status
//
while (1)
{
    if (in_8 (io + 0x07) & 0x80) // ISR
    {
        //
        // Il registro ISR segnala l'azzeramento avvenuto e
        // si azzerava l'indicatore relativo, senza
        // interferire con gli altri.
        //
        out_8 ((io + 0x07), 0x80); // ISR
        break;
    }
}
//
// Si procede con la configurazione necessaria per poter
// leggere dei dati dalla memoria interna, impostando il
// registro DCR (0x0E).
//
// Data configuration register (DCR)
// -----.
// | - |FT1|FT0|ARM| LS|LAS|BOS|WTS|
// |-----|
// | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0x48
// `-----'
//      \_____/ : | : : :
//          | : | : : Byte DMA transfer
//          | : | : :

```

```
//      |      :      |      :      Little endian byte order
//      |      :      |      :
//      |      :      |      Dual 16 bit DMA mode
//      |      :      |
//      |      :      Loopback OFF (normal operation)
//      |      :
//      |      All packets removed from
//      |      Buffer Ring under program control
//      |
//      FIFO threshold 8 bytes
//
out_8 ((io + 0x0E), 0x48);
//
// Azzera i registri RBCRn (0x0A e 0x0B) che rappresentano
// la quantità di byte da trasferire.
//
out_8 ((io + 0x0A), 0x00);          // RBCR0
out_8 ((io + 0x0B), 0x00);          // RBCR1
//
// Disabilita tutti gli impulsi di interruzione, azzerando
// la maschera nel registro IMR (0x0F).
//
out_8 ((io + 0x0F), 0x00);          // IMR
//
// Azzera nuovamente tutti gli indicatori del registro
// ISR (0x07) assegnando tutti i bit a uno.
//
out_8 ((io + 0x07), 0xFF);          // ISR
//
// Configura il registro RCR (0x0C) in modo da rimanere in
// modalità monitor (per non accumulare i pacchetti
// eventualmente ricevuti).
//
// Receive configuration register (RCR)
```

```

// .-----.
// | - | - |MON|PRO| AM| AB| AR|SEP|
// |-----|
// | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0x20
// `-----'
//           | : : : : :
//           | : : : :   Packets with receive errors
//           | : : : :   are rejected
//           | : : : :
//           | : : : :   Packets with fewer than 64
//           | : : : :   bytes rejected
//           | : : : :
//           | : :   Packets with broadcast destination
//           | : :   rejected accepted
//           | : :
//           | :   Packets with multicast destination
//           | :   address not checked
//           | :
//           |   Physical address of node must match the
//           |   station address
//           |
//           Monitor mode: packets checked but not buffered
//           to memory
//
out_8 ((io + 0x0C), 0x20);           // RCR
//
// Configura il registro TCR (0x0D) in modo da rimanere in
// modalità «loopback» (per non trasmettere pacchetti).
//
// Transmit configuration register (TCR)
// .-----.
// | - | - | - |OFST|ATD|LB1|LB0|CRC|
// |-----|
// | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0x02

```

```

// '-----'
//          \_____/ :
//          |      CRC appended by the
//          |      transmitter
//          |
//          Internal loopback (mode 1)
//
out_8 ((io + 0x0D), 0x02);          // TCR
//
// Si procede finalmente indicando la quantità di byte da
// leggere dalla memoria. Dato che l'indirizzo Ethernet
// appare doppio, vanno letti 12 byte. Si devono impostare i
// registri RBCRn (0x0A e 0x0B), suddividendo la quantità
// nei due ottetti.
//
out_8 ((io + 0x0A), 12);          // RBCR0
out_8 ((io + 0x0B), 12 >> 8));  // RBCR1
//
// Si imposta la posizione iniziale di lettura a zero, da
// dove inizia l'informazione da leggere. Si impostano i
// registri RSARn (0x08 e 0x09).
//
out_8 ((io + 0x08), 0);          // RSAR0
out_8 ((io + 0x09), 0);          // RSAR1
//
// Si imposta il registro CR (0x00) per iniziare la lettura.
//
// Command register (CR)
// .-----
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|
// | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
// '-----'
// \_____/ \_____/ |

```

```

//      |      |      START
//      |      Read
//      |
//      Register
//      page 0
//
out_8 ((io + 0x00), 0x0A);                                // CR
//
// Attraverso la porta di comunicazione 0x10 si leggono i
// dati richiesti, salvandoli opportunamente.
//
for (i = 0; i < 12; i++)
{
    sa_prom[i] = in_8 (io + 0x10);                        // DATA
}
//
// Si trasferiscono i dati utili nell'array che contiene
// l'indirizzo fisico dell'interfaccia.
//
par[0] = sa_prom[0];
par[1] = sa_prom[2];
par[2] = sa_prom[4];
par[3] = sa_prom[6];
par[4] = sa_prom[8];
par[5] = sa_prom[10];
//
// Dopo l'azzeramento iniziale e l'acquisizione
// dell'indirizzo fisico, si procede finalmente con la
// sequenza di inizializzazione finale. Di nuovo si ferma
// tutto con il registro CR (0x00).
//
// Command register (CR)
// .-----.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|

```

```

// |-----|
// | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
// `-----'
// \_____/ \_____/ |
// | | | STOP
// | | |
// | Abort/complete
// | remote DMA
// |
// Register
// page 0
//
out_8 ((io + 0x00), 0x21); // CR
//
// A questo punto, il registro ISR (0x07) potrebbe riportare
// lo stato di azzeramento, oppure quello di completamento
// del trasferimento dati tra la memoria interna e quella
// esterna. Tuttavia, la sua interrogazione non dovrebbe
// essere necessaria.
//
// Si imposta il registro DCR (0x0E) per una modalità di
// funzionamento normale.
//
// Data configuration register (DCR)
// .-----.
// | - |FT1|FT0|ARM| LS|LAS|BOS|WTS|
// |-----|
// | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0x48
// `-----'
// \_____/ : | : :
// | : | : : Byte DMA transfer
// | : | : :
// | : | : Little endian byte order
// | : | :

```



```

//      |      :      |      Dual 16 bit DMA mode
//      |      :      |
//      |      :      Loopback OFF (normal operation)
//      |      :
//      |      Send Command non executed: all packets
//      |      removed from Buffer Ring under program
//      |      control
//      |
//      FIFO threshold 8 bytes
//
out_8 ((io + 0x0E), 0x48);          // DCR
//
// Azzera i registri RBCRn (0x0A e 0x0B).
//
out_8 ((io + 0x0A), 0);           // RBCR0
out_8 ((io + 0x0B), 0);           // RBCR1
//
// Imposta il registro RCR (0x0C) per un funzionamento
// normale.
//
// Receive configuration register (RCR)
// .------.
// | - | - |MON|PRO| AM| AB| AR|SEP|
// |-----|
// | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0x04
// `-----'
//      :      :      :      |      :      :
//      :      :      :      |      :      Packets with receive errors
//      :      :      :      |      :      are rejected
//      :      :      :      |      :
//      :      :      :      |      Packets with less than 64 bytes
//      :      :      :      |      rejected
//      :      :      :      |
//      :      :      :      Packets with broadcast destination

```

```

//      :      :      :      address accepted
//      :      :      :
//      :      :      Packets with multicast destination
//      :      :      address not checked
//      :      :
//      :      Physical address of node must match the
//      :      station address
//      :
//      Packets buffered to memory
//
out_8 ((io + 0x0C), 0x04);          // RCR
//
// Con il registro TPSR (0x04) configura la «pagina»
// iniziale dell'area di memoria usata per collocare i
// pacchetti da trasmettere: 0x40 (pari all'indirizzo
// 0x4000).
//
out_8 ((io + 0x04), 0x40);        // TPSR
//
// Per il momento si lascia il trasmettitore in modalità
// «loopback».
//
// Transmit configuration register (TCR)
// .------.
// | - | - | - |OFST|ATD|LB1|LB0|CRC|
// |-----|
// | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0x02
// |-----'
//
//          \_____/ :
//          |      CRC appended by the
//          |      transmitter
//          |
//          Internal loopback (mode 1)
//

```

```

out_8 ((io + 0x0C), 0x02);           // TCR
//
// Imposta la pagina iniziale per la coda di ricezione dei
// pacchetti, con il registro PSTART (0x01). Si indica 0x46,
// pari all'indirizzo 0x4600.
//
out_8 ((io + 0x01), 0x46);         // PSTART
//
// Imposta l'indice di lettura, attraverso il registro BNRy
// (0x03). All'inizio questo indice coincide con la pagina
// iniziale della coda di ricezione.
//
out_8 ((io + 0x03), 0x46);         // BNRy
//
// Imposta la pagina di memoria finale della coda di
// ricezione (precisamente si tratta della pagina successiva
// alla fine della coda), attraverso il registro PSTOP
// (0x02). Viene indicata la pagina 0xC0, pari all'indirizzo
// 0xC000.
//
out_8 ((io + 0x02), 0xC0);         // PSTOP
//
// Utilizza il registro CR (0x00) per passare alla seconda
// pagina di registri.
//
// Command register (CR)
// .------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|
// | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0x61
// `-----'
// \_____/ \_____/ |
//   |           |   STOP
//   |           |

```

```
//      |      Abort/complete
//      |      remote DMA
//      |
//      Register
//      page 1
//
out_8 ((io + 0x00), 0x61);          // CR
//
// Imposta i registri PARn e MARn con l'indirizzo fisico e
// l'indirizzo multicast. I registri PARn vanno da 0x01 a
// 0x06; i registri MARn vanno da 0x08 a 0x0F.
//
out_8 ((io + 0x01), par[0]);      // PAR0
out_8 ((io + 0x02), par[1]);      // PAR1
out_8 ((io + 0x03), par[2]);      // PAR2
out_8 ((io + 0x04), par[3]);      // PAR3
out_8 ((io + 0x05), par[4]);      // PAR4
out_8 ((io + 0x06), par[5]);      // PAR5
//
out_8 ((io + 0x08), 0);           // MAR0
out_8 ((io + 0x09), 0);           // MAR1
out_8 ((io + 0x0A), 0);           // MAR2
out_8 ((io + 0x0B), 0);           // MAR3
out_8 ((io + 0x0C), 0);           // MAR4
out_8 ((io + 0x0D), 0);           // MAR5
out_8 ((io + 0x0E), 0);           // MAR6
out_8 ((io + 0x0F), 0);           // MAR7
//
// Imposta l'indice di scrittura nella coda di ricezione,
// per i pacchetti ricevuti. Per questo si usa il registro
// CURR (0x07). Viene indicata la pagina di memoria iniziale
// della coda di ricezione.
//
out_8 ((io + 0x07), 0x46);        // CURR
```

```

//
// Attraverso il registro CR (0x00) viene ripristinata la
// prima pagina di registri e viene attivata l'interfaccia.
//
// Command register (CR)
// .------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|
// | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0x22
// `-----'
// \_____/ \_____/ |
//      |           |          START
//      |           |
//      |           Abort/complete
//      |           remote DMA
//      |
//      Register
//      page 0
//
out_8 ((io + 0x00), 0x22);          // CR
//
// Si azzerano tutti gli indicatori del registro ISR (0x07).
//
out_8 ((io + 0x07), 0xFF);        // ISR
//
// A questo punto bisogna decidere se si vogliono ottenere
// dei segnali di interruzione, o meno. Nel caso si vogliono
// gestire le interruzioni, potrebbe essere conveniente
// abilitare quelle generate da un errore di trasmissione
// (0x08), dalla conclusione corretta di una trasmissione
// (0x02) e dalla ricezione corretta di un pacchetto (0x01).
// In tal caso, la maschera da adottare dovrebbe essere
// 0x0B.
//

```

```

// Tuttavia, volendo interrogare l'interfaccia in modo
// regolare, senza utilizzare le interruzioni, si può
// azzerare la maschera del registro IMR (0x0F), come in
// questo caso.
//
out_8 ((io + 0x0F), 0x00); // IMR
//
// Azzerare il registro TCR (0x0D) per ottenere una modalità
// normale del funzionamento del trasmettitore (non più in
// «loopback»).
//
// Transmit configuration register (TCR)
// .-----.
// | - | - | - |OFST|ATD|LB1|LB0|CRC|
// |-----|
// | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x00
// |-----'
//
out_8 ((io + 0x0D), 0x00); // TCR
//
// Fine del procedimento di inizializzazione.
//
return (0);

```

83.11.8 Trasmissione di un pacchetto



Per la trasmissione è sufficiente riservare un'area di memoria interna pari alla dimensione massima che un pacchetto singolo può occupare. In pratica si tratta di 1536 byte, pari a sei blocchi (pagine) da 256 byte. Negli esempi apparsi in precedenza, lo spazio destinato alla trasmissione è stato collocato tra 4000_{16} e $45FF_{16}$, estremi inclusi.

Avendo già impostato l'interfaccia come descritto nella sezione precedente, per poter trasmettere un pacchetto occorre scriverlo nell'area di memoria interna prevista e poi richiederne la trasmissione. Durante questa fase può succedere di scoprire che il trasmettitore sia già impegnato, per cui conviene rinunciare e riprovare in un momento successivo. D'altra parte, in un momento successivo alla trasmissione occorre verificare che non si sia presentato un errore nella trasmissione stessa (eventualmente a seguito della ricezione di un'interruzione, se abilitata).

Nel listato successivo, *buffer* è un puntatore a un'area di memoria dell'elaboratore, contenente il pacchetto da trasmettere, mentre *size* contiene la dimensione complessiva in byte del pacchetto stesso. La variabile *io* rappresenta sempre la porta di I/O iniziale, per accedere all'interfaccia.

```

int      i;
int      status;
uint8_t *b = buffer;
//
// Si legge il registro CR (0x00) per determinare se
// l'interfaccia è già in fase di trasmissione. Nel caso,
// occorre rinunciare per riprovare in un momento
// successivo.
//
// Command register (CR)
// -----.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|
// | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0x26
// '-----'
//  \_____/ \_____/ | |
//  | | | Start
//  | | | Transmit packet

```

```

//      |          Abort/complete
//      |          remote DMA
//      Register
//      page 0
//
status = in_8 (io + 0x00);                                // CR
if (status == 0x26)
{
    errset (EBUSY);
    return (-1);
}
//
// Si immette la dimensione da trasmettere nei registri
// RBCRn (0x0A e 0x0B).
//
out_8 ((io + 0x0A), (size & 0xFF));                      // RBCR0
out_8 ((io + 0x0B), (size >> 8));                       // RBCR1
//
// Si indica la posizione iniziale della memoria interna in
// cui depositare la copia del pacchetto da trasmettere
// (0x4000). Si utilizzano per questo i registri RSARn
// (0x08 e 0x09).
//
out_8 ((io + 0x08), 0x00);                               // RSAR0
out_8 ((io + 0x09), 0x40);                               // RSAR1
//
// Si dichiara l'intenzione di procedere con una scrittura
// nella memoria interna.
//
// Command register (CR)
// .------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|
// | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0x12

```



```

// '-----'
// \_____/ \_____/ |
// | | | Start
// | | |
// | Write
// Register
// page 0
//
out_8 ((io + 0x00), 0x12); // CR
//
// Si procede con il trasferimento della copia del
// pacchetto, attraverso la scrittura della porta 0x10.
//
for (i = 0; i < size; i++)
{
    out_8 ((io + 0x10), b[i]); // DATA
}
//
// Si attende che il registro ISR (0x07) confermi il
// completamento dell'operazione.
//
// Interrupt status register (ISR)
// .-----
// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-----|
// | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
// '-----'
// |
// Remote DMA complete
//
while (1)
{
    if (in_8 (io + 0x07) & 0x40) // ISR
    {

```

```

//
// Il registro ISR dà la conferma e se ne azzera
// l'indicatore relativo, senza interferire con gli
// altri.
//
out_8 ((io + 0x07), 0x40); // ISR
break;
}
}
//
// Si dichiara il blocco di memoria interna (pagina) in cui
// è contenuto il pacchetto da inviare. Si imposta il
// registro TPSR (0x04) con il valore 0x40, corrispondente
// al blocco che inizia all'indirizzo 0x4000.
//
out_8 (io + 0x04, 0x40); // TPSR
//
// Si dichiara la quantità di byte da trasmettere,
// utilizzando i registri TBCRn (0x05 e 0x06).
//
out_8 ((io + 0x05), (size & 0xFF)); // TBCR0
out_8 ((io + 0x06), (size >> 8)); // TBCR1
//
// Finalmente si trasmette il pacchetto.
//
// Command register (CR)
// .------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|
// | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0x26
// `-----'
// \_____/ \_____/ | |
// | | | | | Start
// | | | | | Transmit packet

```

```

//      |          Abort/complete remote DMA
//      Register
//      page 0
//
out_8 ((io + 0x00), 0x26);                      // CR
//
// Si attende che il pacchetto sia stato trasmesso o che sia
// riportato un errore.
//
// Interrupt status register (ISR)
// .------.
// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-----|
// | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
// `-----'
//
//          |          |
//          |          Packet transmitted with no
//          |          errors
//          |          Transmit error
//
while (1)
{
    if (in_8 (io + 0x07) & 0x0A)                // ISR
    {
        //
        // Azzeramento degli indicatori previsti.
        //
        out_8 ((io + 0x07), 0x0A);                // ISR
        break;
    }
}
//
// Si verifica nel registro TSR (0x04) l'esito della
// trasmissione.

```

```

//
// Transmit status (TSR)
// .------.
// |OWC|CDH| FU|CRS|ABT|COL| - |PTX|
// |-----|
// | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0x38
// `-----'
//
//           |   |   |
//           |   |   Transmit aborted
//           |   Carrier sense lost
//           FIFO underrun
//
status = in_8 (io + NE2K_TSR);
if (status & 0x38)
{
    errset (EIO);
    return (-1);
}
//
// Fine
//
return (0);

```

83.11.9 Ricezione

«

Alla ricezione dei pacchetti (trame) provvede l'interfaccia, ammesso di avere configurato tutto opportunamente, come mostrato in precedenza, sapendo che il registro **CURR** indica il blocco (la pagina) della memoria interna in cui va collocato il prossimo pacchetto ricevuto. Per il prelievo di questi dati dalla memoria interna, si utilizza il registro **BNRY**, il quale rappresenta il blocco di memoria ancora da leggere. Sapendo che inizialmente i registri **BNRY** e **CURR** puntano

entrambi al blocco iniziale di memoria interna destinato ad accogliere i dati ricevuti, il valore contenuto nel registro *BNRY* non può superare *CURR*. Quando però la ricezione procede velocemente, più di quanto si provveda a estrarre i pacchetti, il registro *CURR* può raggiungere di nuovo *BNRY*, ma in tal caso si ottiene uno straripamento che deve essere gestito in qualche modo.

Secondo l'organizzazione prevista in precedenza, la porzione di memoria interna destinata alla ricezione dei pacchetti va da 4600_{16} a $BFFF_{16}$, inclusi.

Nell'esempio del listato seguente, come già in quelli precedenti, la variabile *io* rappresenta la porta di I/O iniziale, per accedere all'interfaccia; inoltre, *destination* è un puntatore all'area di memoria in cui va collocato un pacchetto; tale puntatore si ottiene attraverso una funzione, denominata *new_frame()*.

```

int    i;
int    bytes;
int    curr;
int    bnry;
int    next;
int    frame_status;
int    frame_size;
int    status;
char *destination;
//
// Si seleziona la seconda pagina di registri, per poter
// accedere poi al registro CURR.
//
// Command register (CR) 0x00
// .-----
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|

```

```

// | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0x62
// '-----'
// \_____/ \_____/ |
// | | | START
// | | |
// | | | Abort/complete remote DMA
// |
// Register page 1
//
out_8 ((io + 0x00), 0x62); // CR
//
// Legge la posizione corrente dell'indice di scrittura
// CURR (0x07).
//
curr = in_8 (io + 0x07); // CURR
//
// Si seleziona nuovamente la prima pagina di registri.
//
// Command register (CR) 0x00
// .-----
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|
// | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0x22
// '-----'
// \_____/ \_____/ |
// | | | START
// | | |
// | | | Abort/complete remote DMA
// |
// Register page 0
//
out_8 ((io + 0x00), 0x22); // CR
//
// Si legge il valore contenuto nel registro BNRV (0x03).

```

```

//
bnry = in_8 (io + 0x03);           // BNRy
//
// Si parte dal presupposto che ci sia almeno un pacchetto
// da leggere; pertanto, se anche i registri CURR e BNRy
// fossero uguali, significherebbe solo che tutta la memoria
// interna destinata alla ricezione, richiede di essere letta.
//
// Si passa al prelievo di tutti i pacchetti pronti per il
// prelievo nell'area di memoria interna.
//
while (1)
{
    //
    // Trova un posto dove mettere un pacchetto ricevuto.
    //
    destination = new_frame ();
    //
    // Ci si prepara a leggere i primi quattro byte,
    // iniziando dal blocco di memoria a cui si riferisce la
    // variabile 'bnry'.
    //
    out_8 ((io + NE2K_RBCR0), 4);
    out_8 ((io + NE2K_RBCR1), 0);
    //
    out_8 ((io + NE2K_RSAR0), 0); // Deve essere zero!
    out_8 ((io + NE2K_RSAR1), bnry);
    //
    // Si predispose il registro CR (0x00) per la lettura
    // della memoria interna.
    //
    // Command register (CR)
    // .-----
    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|

```

```

// |-----|
// | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
// `-----'
// \_____/ \_____/ |
// | | | START
// | | |
// | | | Read
// |
// Register page 0
//
out_8 (io + 0x00, 0x1A); // CR
//
// Si leggono e si salvano le informazioni ottenute. La
// lettura si esegue dalla porta di comunicazione
// destinata alla trasferimento di dati con la memoria
// interna.
//
frame_status = in_8 (io + 0x10); // frame status
next = in_8 (io + 0x10); // next frame
frame_size = in_8 (io + 0x10); // frame size
// low
frame_size += (in_8 (io + 0x10) * 256); // frame size
// high

//
// Si attende che il registro ISR (0x07) confermi il
// completamento dell'operazione di lettura.
//
// Interrupt status register (ISR)
// .------.
// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-----|
// | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
// `-----'
// |

```



```
//      Remote DMA complete
//
while (1)
{
    if (in_8 (io + 0x07) & 0x40)                // ISR
        {
            //
            // Azzeramento degli indicatori previsti.
            //
            out_8 ((io + 0x07), 0x40);           // ISR
            break;
        }
}
//
// Adesso si sa quanto è grande il pacchetto da leggere,
// ma si deve ripetere la lettura a cominciare dai primi
// quattro byte di intestazione, anche se questi non
// fanno parte del pacchetto (la trama Ethernet).
//
if (next > bnry)
{
    bytes = frame_size;
}
//
if (next < bnry)
{
    //
    // Se il pacchetto successivo si trova in un blocco
    // antecedente alla posizione iniziale di lettura,
    // significa che quello attuale si estende oltre la
    // fine dell'area di memoria (riprendendo all'inizio
    // della stessa), pertanto in questo caso occorre
    // procedere con due fasi di lettura. Il valore 0xC0
    // rappresenta il blocco di memoria successivo alla
```

```

// conclusione dell'area di memoria destinata alla
// ricezione. La quantità di blocchi da leggere
// viene moltiplicata per 256, essendo questa la
// dimensione di un blocco.
//
bytes = ((0xC0 - bnry) * 256);
bytes = min (bytes, frame_size);
}
//
// Prima fase di lettura. Si ricorda che la lettura
// riparte dall'inizio, pertanto vanno poi eliminati i
// primi quattro byte di intestazione.
//
out_8 ((io + NE2K_RBCR0), bytes & 0xFF);
out_8 ((io + NE2K_RBCR1), bytes >> 8);
//
out_8 ((io + NE2K_RSAR0), 0);      // MUST be zero. :-(
out_8 ((io + NE2K_RSAR1), bnry);
//
// Si imposta il registro CR (0x00) per la lettura.
//
// Command register (CR)
// .------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|
// | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
// |-----'
// \_____/ \_____/ |
// |           |      START
// |           |
// |           |      Read
// |           |
// |           |
// |           |      Register page 0
//

```

```

out_8 (io + 0x00, 0x0A);           // CR
//
// Legge e scarta i primi quattro byte di intestazione.
//
in_8 (io + 0x10);                 // DATA
in_8 (io + 0x10);                 // DATA
in_8 (io + 0x10);                 // DATA
in_8 (io + 0x10);                 // DATA
//
// Riduce la quantità di byte ancora da leggere di
// conseguenza.
//
bytes -= 4;
//
// Passa alla lettura del pacchetto.
//
i = 0;
for (; bytes > 0; i++, bytes--)
{
    destination[i] = in_8 (io + 0x10); // DATA
}
//
// Si attende la conclusione dell'operazione,
// controllando l'esito nel registro ISR (0x07).
//
// Interrupt status register (ISR)
// -----
// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-----|
// | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
// '-----'
//
// |
// Remote DMA complete
//

```

```
while (1)
{
    if (in_8 (io + 0x07) & 0x40) // ISR
    {
        //
        // Azzeramento degli indicatori previsti.
        //
        out_8 ((io + 0x07), 0x40); // ISR
        break;
    }
}
//
if (next < bnry)
{
    //
    // Dato che il pacchetto successivo si trova in un
    // blocco di memoria antecedente all'inizio di
    // quello letto, potrebbe esserci una seconda parte
    // da leggere.
    // Se il calcolo successivo da un risultato maggiore
    // di zero, c'è effettivamente ancora da leggere, a
    // partire dal blocco di memoria 0x46.
    //
    bytes = frame_size - ((0xC0 - bnry) * 256);
}
//
if (bytes > 0)
{
    //
    out_8 ((io + NE2K_RBCR0), bytes & 0xFF);
    out_8 ((io + NE2K_RBCR1), bytes >> 8);
    //
    out_8 ((io + NE2K_RSAR0), 0);
    out_8 ((io + NE2K_RSAR1), 0x46);
}
```

```

//
// Command register (CR)
// .------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----|
// | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
// `-----'
// \_____/ \_____/ |
// | | | START
// | | |
// | | | Read
// | | |
// Register page 0
//
out_8 (io + 0x00, 0x0A); // CR
//
for (; bytes > 0; i++, bytes--)
{
    destination[i] = in_8 (io + 0x10); // DATA
}
//
// Interrupt status register (ISR)
// .------.
// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-----|
// | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
// `-----'
// |
// Remote DMA complete
//
while (1)
{
    if (in_8 (io + 0x07) & 0x40) // ISR
        {

```

```

        //
        // Azzeramento degli indicatori previsti.
        //
        out_8 ((io + 0x07), 0x40);           // ISR
        break;
    }
}
//
// Essendosi conclusa la lettura del pacchetto, si aggiorna B
// 0x03.
//
bnry = next;
out_8 (io + 0x03, bnry);                 // BNRy
//
// Se il valore attuale di BNRy coincide con quello di
// CURR, il ciclo termina.
//
if (bnry == curr)
{
    //
    // Fine.
    //
    return;
}
}

```

83.12 IPv4 in una rete Ethernet

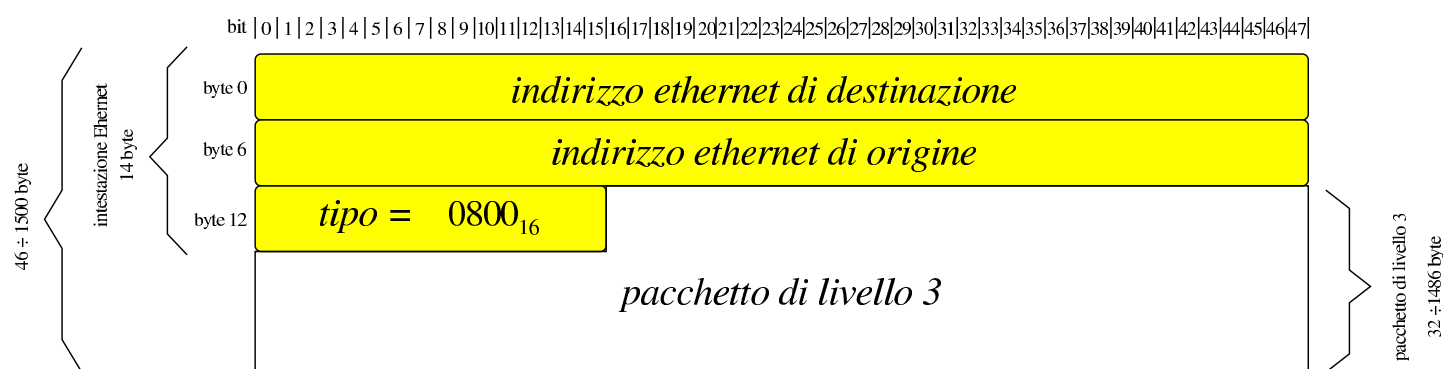


Il protocollo IPv4 è ormai superato, ma rimane ancora in uso e, per la sua semplicità, si presta meglio a un primo approccio alla gestione dei protocolli TCP/IP.

Negli schemi delle figure che appaiono in questa sezione, si intende che i dati siano ordinati secondo il così detto *network byte order*, ossia come sequenza di byte, da sinistra verso destra; per la stessa ragione, i bit sono numerati partendo dal bit più significativo in giù. Per converso, negli esempi di strutture in linguaggio C, usati per rappresentare i dati contenuti nei pacchetti dei protocolli, si intende di operare in un'architettura di tipo *little endian*

I pacchetti del protocollo Ethernet hanno l'intestazione descritta dalla figura successiva. Ciò che segue tale intestazione è poi il pacchetto dal punto di vista del protocollo di rete.

Figura 83.129. PDU di livello 2, nelle reti Ethernet.



83.12.1 ARP

Il protocollo ARP permette di risalire all'indirizzo fisico, partendo da quello di rete. In una rete Ethernet, gli indirizzi fisici occupano 48 bit, ovvero 6 byte. I pacchetti del protocollo ARP hanno la struttura evidenziata dalla figura successiva.

Figura 83.130. Struttura di un pacchetto del protocollo ARP, relativo a una rete fisica Ethernet e al protocollo IPv4.

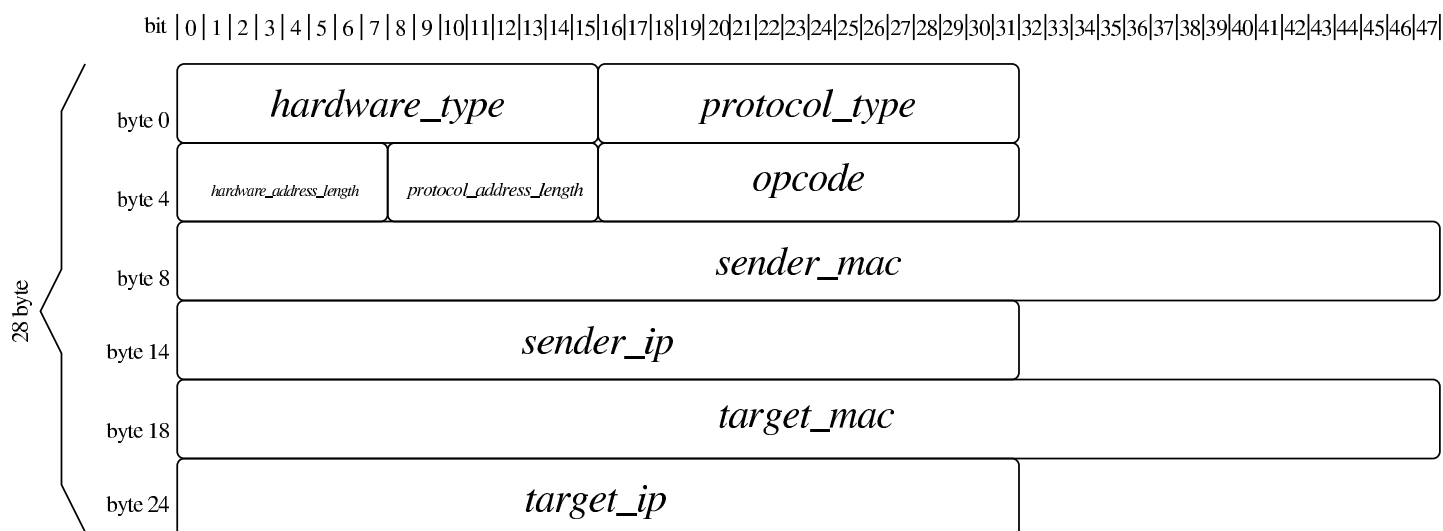


Tabella 83.131. Campi di un pacchetto del protocollo ARP, relativo a una rete fisica Ethernet e al protocollo IPv4.

Campo	Lunghezza	Descrizione
<i>hardware_type</i>	16 bit	Descrive il tipo di rete fisica; per una rete Ethernet si usa il codice 0001 ₁₆ .
<i>protocol_type</i>	16 bit	Descrive il tipo di indirizzamento a livello 3 del modello ISO-OSI; per il protocollo IPv4 si usa il codice 0800 ₁₆ .
<i>hardware_address_length</i>	8 bit	Dichiara la dimensione dell'indirizzo hardware in ottetti (byte); nel caso del protocollo Ethernet, l'indirizzo occupa 6 ottetti.
<i>protocol_address_length</i>	8 bit	Dichiara la dimensione dell'indirizzo di rete in ottetti (byte); nel caso del protocollo IPv4, l'indirizzo occupa 4 ottetti.

Campo	Lunghezza	Descrizione
<i>opcode</i>	16 bit	Dichiara il tipo di operazione richiesta; il valore 0001_{16} indica la richiesta di conoscere l'indirizzo fisico corrispondente a un certo indirizzo di rete; il valore 0002_{16} indica la risposta con l'informazione richiesta.
<i>sender_mac</i>	variabile	Contiene l'indirizzo fisico di origine.
<i>sender_ip</i>	variabile	Contiene l'indirizzo di rete di origine.
<i>target_mac</i>	variabile	Contiene l'indirizzo fisico di destinazione.
<i>target_ip</i>	variabile	Contiene l'indirizzo di rete di destinazione.

Listato 83.132. Esempio di struttura per descrivere un pacchetto del protocollo ARP, destinato a essere usato in una rete Ethernet. L'ultimo campo, denominato qui *filler*, serve a far sì che il pacchetto raggiunga almeno la dimensione minima richiesta dal protocollo Ethernet.

```
typedef struct {
    uint16_t hardware_type;
    uint16_t protocol_type;
    uint8_t  hardware_address_length;           // 6 byte
    uint8_t  protocol_address_length;         // 4 byte
    uint16_t opcode;
    uint8_t  sender_mac[6];
    uint32_t sender_ip; // Network byte order.
    uint8_t  target_mac[6];
    uint32_t target_ip; // Network byte order.
    uint8_t  filler[4];
} __attribute__((packed)) arp_packet_t;
```

L'utilizzo più comune del protocollo prevede l'invio di un pacchetto di richiesta, per conoscere l'indirizzo fisico di un certo indirizzo di rete, per il quale ci si aspetta di ottenere un pacchetto di risposta, contenente l'informazione richiesta, dal nodo che utilizza effettivamente quell'indirizzo di rete cercato.

Va però osservato che il protocollo ARP serve a collegare il protocollo fisico con quello di rete; pertanto, per l'analisi dei pacchetti ARP occorre considerare anche il loro involucro a livello fisico.

A titolo di esempio, si parte dalla situazione schematizzata dalla figura successiva, dove il nodo «A» cerca di contattare il nodo «B», ma per farlo deve conoscerne l'indirizzo fisico, attraverso l'ausilio del protocollo ARP.

Figura 83.133. Situazione ipotetica di due nodi che utilizzano il protocollo ARP.

IPv4: 172.21.11.1

IPv4: 172.21.11.2

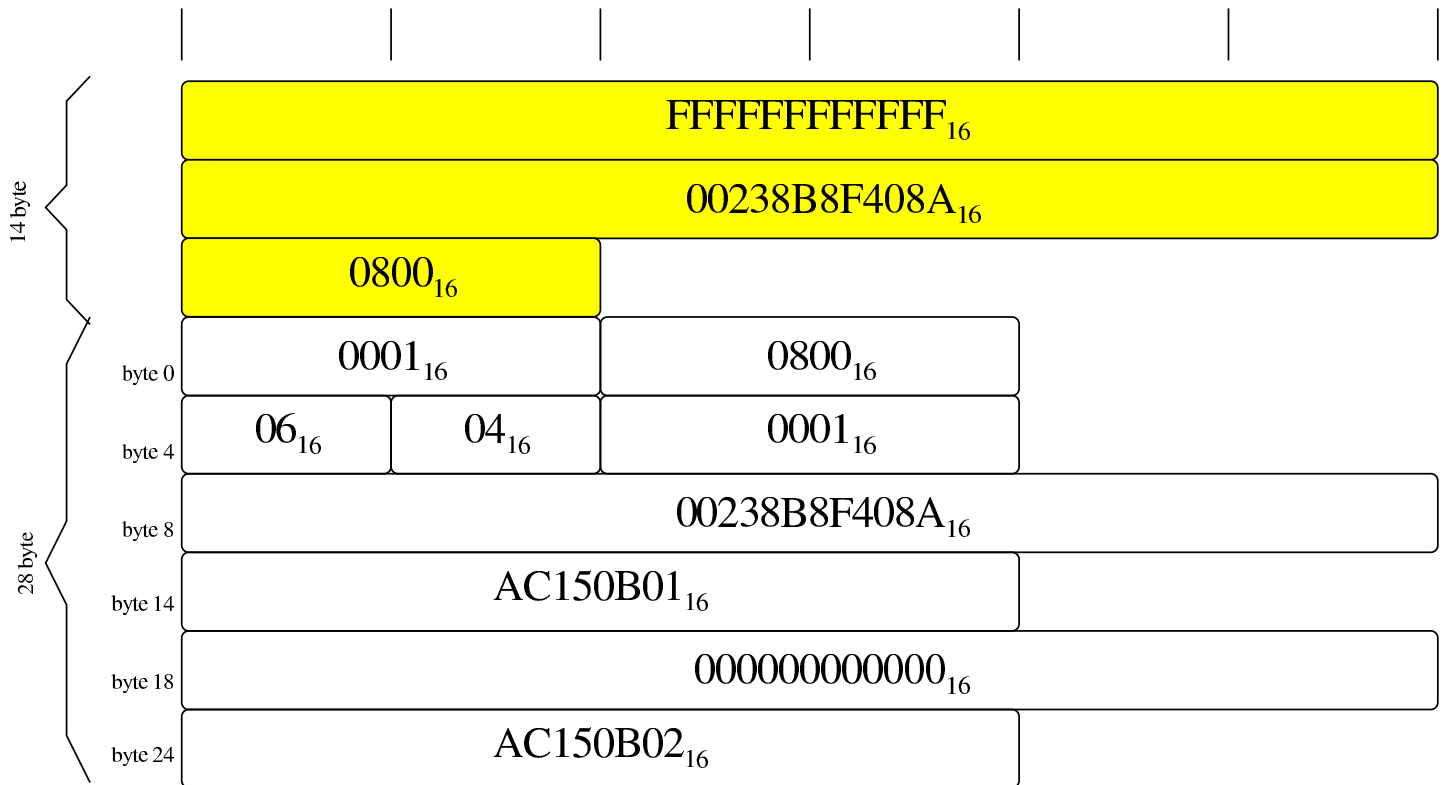
Ethernet: 00:23:8b:8f:40:8a

Ethernet: 00:05:5d:4c:4d:d9



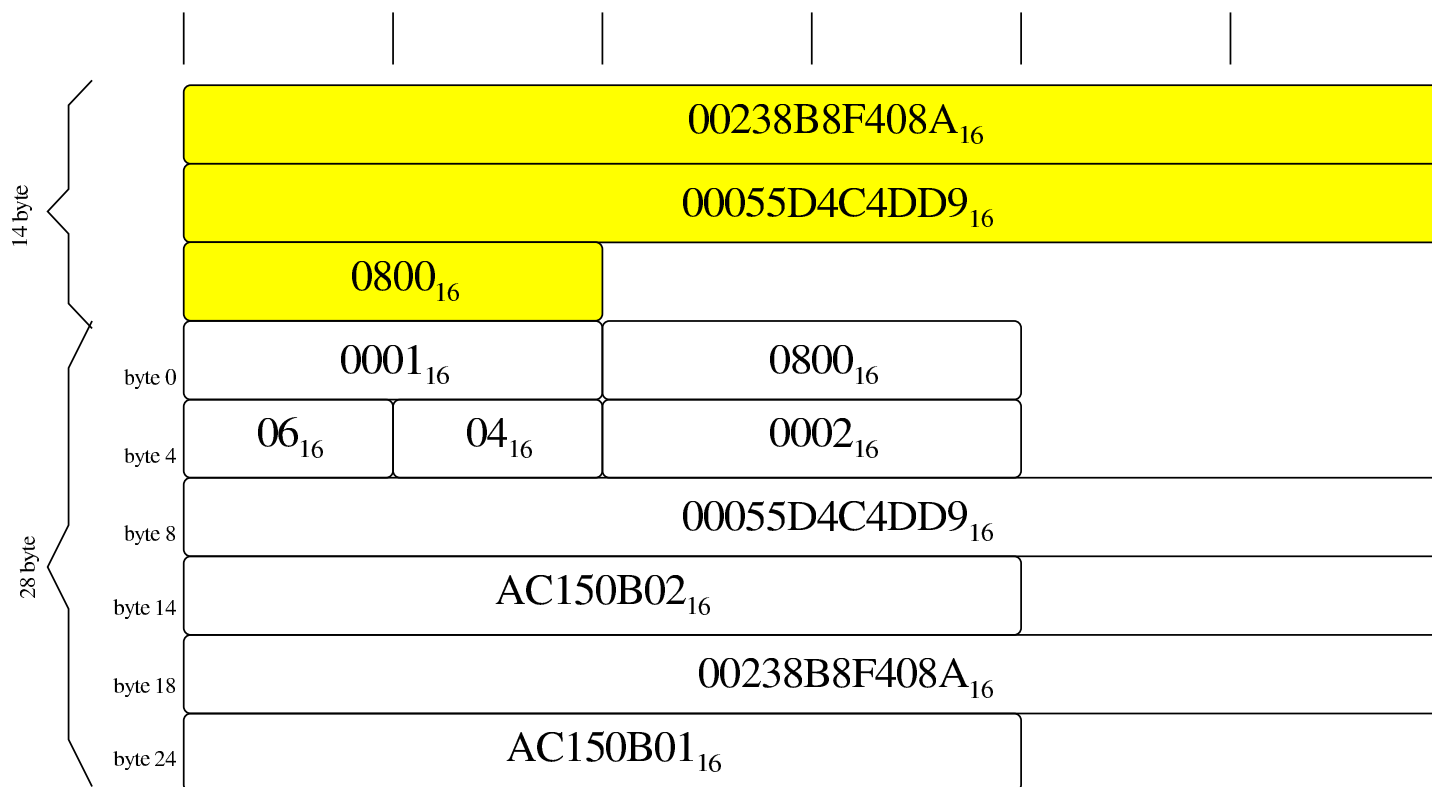
Il nodo «A» invia un pacchetto di richiesta nella rete fisica locale. Questo pacchetto deve essere diretto a tutti i nodi fisici raggiungibili, pertanto è contenuto in un pacchetto Ethernet «circolare», ovvero *broadcast*.

Figura 83.134. Pacchetto di richiesta ARP per conoscere l'indirizzo fisico del nodo «B» (172.21.11.2). I primi 14 byte rappresentano l'involucro Ethernet, nel quale si può osservare che la destinazione è indefinita, utilizzando un indirizzo *broadcast* (ff:ff:ff:ff:ff:ff).



Quando il nodo «B» intercetta il pacchetto di richiesta ARP, nota che l'indirizzo IPv4 contenuto riguarda la sua interfaccia di rete, pertanto risponde con un altro pacchetto ARP, ma in tal caso la destinazione è precisa, perché conosciuta dal pacchetto di richiesta.

Figura 83.135. Pacchetto di risposta ARP per comunicare l'indirizzo fisico del nodo «B» (172.21.11.2) al nodo «A» (172.21.11.1). I primi 14 byte rappresentano l'involucro Ethernet.



83.12.2 IPv4

Il protocollo IPv4 si colloca al primo dei livelli interessati dal TCP/IP, mentre nel modello ISO/OSI si tratta del terzo livello, essendo un protocollo di rete. Il protocollo IP utilizza degli indirizzi propri per individuare i vari nodi con cui avviene la comunicazione; tuttavia, questi indirizzi, a livello di rete, vanno tradotti in indirizzi fisici per raggiungere effettivamente la destinazione, cosa che di norma viene gestita con l'ausilio del protocollo ARP, come descritto nella sezione precedente.

L'intestazione di un pacchetto IPv4 ha delle componenti che possono essere piuttosto complesse; in particolare possono essere previste

delle opzioni che allungano in modo variabile questa intestazione. Tuttavia, qui si presume di non gestire mai tali opzioni e di ignorarle semplicemente se contenute nei pacchetti che si ricevono.

Figura 83.136. Struttura di un pacchetto del protocollo IPv4.

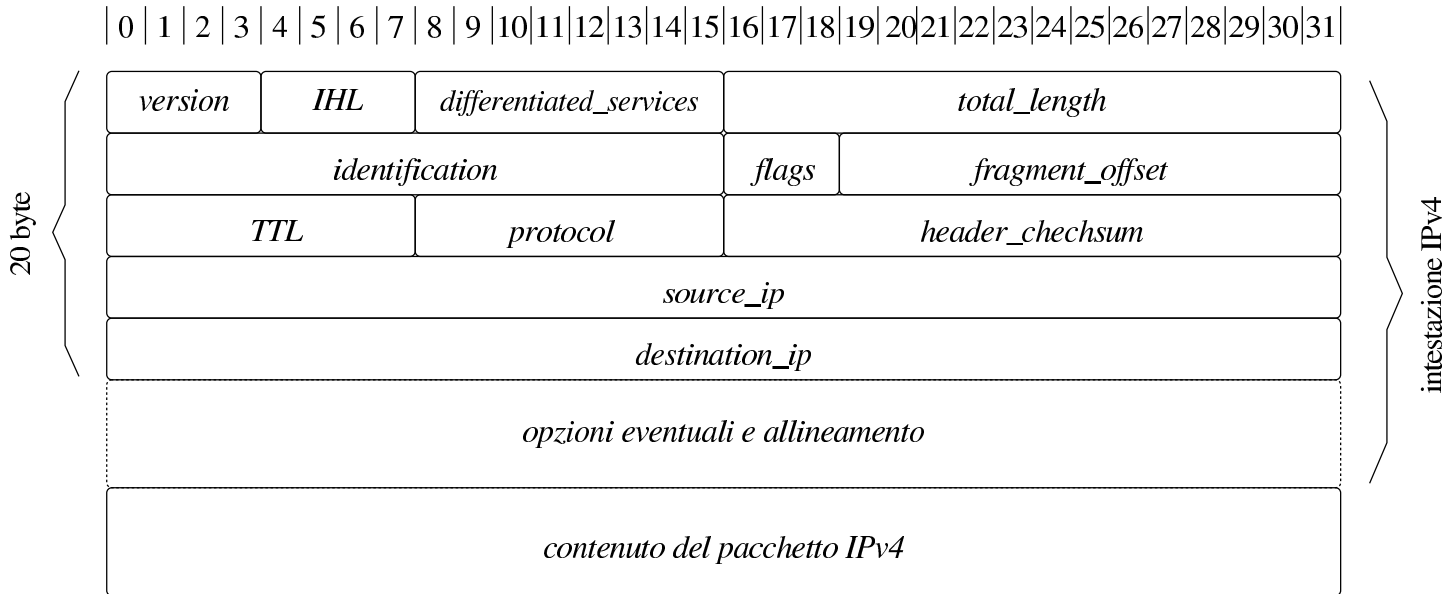


Tabella 83.137. Campi dell'intestazione di un pacchetto del protocollo IPv4. Le opzioni, se presenti, devono occupare uno spazio multiplo di 32 bit, riempiendo eventualmente i bit mancanti con valori a zero.

Campo	Lunghezza	Descrizione
<i>version</i>	4 bit	Descrive la versione del pacchetto a livello del protocollo di rete; per IPv4 si tratta del valore 4.

Campo	Lunghezza	Descrizione
<i>IHL</i> (<i>internet header length</i>)	4 bit	Dichiara la lunghezza dell'intestazione del pacchetto IPv4, in multipli di 32 bit. Dal momento che l'intestazione ha una parte iniziale fissa di 20 ottetti (byte), il valore minimo che può assumere questo campo è pari a 5 e aumenta in presenza di opzioni.
<i>differentiated_services</i>	8 bit	Si tratta di quello che originariamente era noto con la sigla «TOS» (<i>type of service</i>). In generale, per un pacchetto «normale», questo campo ha il valore zero.
<i>total_length</i>	16 bit	Dichiara la lunghezza complessiva del pacchetto IPv4, intestazione inclusa.
<i>identification</i>	16 bit	Numero di identificazione del pacchetto IPv4. Nel caso di pacchetti IP frammentati, questo numero deve rimanere lo stesso per tutti i frammenti che compongono lo stesso pacchetto complessivo.

Campo	Lunghezza	Descrizione
<i>flags</i>	3 bit	Bit indicatori. Nell'ambito di questo gruppo di tre bit, il bit ₀ (ovvero quello meno significativo) è noto con la sigla MF (<i>more fragments</i>) e, se attivo, indica che il pacchetto attuale è composto da altri frammenti successivi. Il bit ₁ è noto con la sigla DF (<i>don't fragment</i>) e, se attivo, indica che il pacchetto non può essere frammentato. Il bit ₂ è riservato.
<i>fragment_offset</i>	13 bit	Questo campo viene usato quando si tratta di un frammento di un pacchetto IPv4 più grande, per indicare che l'inizio del contenuto di questo frammento va collocato a partire dallo scostamento indicato. In un pacchetto non frammentato il contenuto di questo campo è pari a zero.
TTL	8 bit	Il valore di questo campo, noto come <i>time do live</i> , viene stabilito nel momento della sua trasmissione e decrementato di una unità all'attraversamento di ogni router; se questo valore raggiunge lo zero, il pacchetto viene scartato.

Campo	Lunghezza	Descrizione
<i>protocol</i>	8 bit	Il contenuto di un pacchetto IPv4 riguarda un protocollo di trasporto (livello 4 del modello ISO/OSI); questo campo indica di che protocollo si tratta. In particolare: $01_{16} = \text{ICMP};$ $06_{16} = \text{TCP};$ $11_{16} = \text{UDP}.$
<i>header_checksum</i>	16 bit	Codice di controllo calcolato sull'intestazione IPv4, opzioni incluse.
<i>source_ip</i>	32 bit	Indirizzo IPv4 di origine.
<i>destination_ip</i>	32 bit	Indirizzo IPv4 di destinazione.

Listato 83.138. Esempio di struttura in linguaggio C, per descrivere l'intestazione di un pacchetto del protocollo IPv4. A sinistra nella versione *little endian*; a destra in quella *big endian*. Il campo *frag_off* include qui i bit *DF* e *MF*.

<i>little endian</i>	<i>big endian</i>
<pre> struct iphdr { uint16_t ihl : 4, version : 4; uint8_t tos; uint16_t tot_len; uint16_t id; uint16_t frag_off; uint8_t ttl; uint8_t protocol; uint16_t check; uint32_t saddr; uint32_t daddr; }; </pre>	<pre> struct iphdr { uint16_t version : 4, ihl : 4; uint8_t tos; uint16_t tot_len; uint16_t id; uint16_t frag_off; uint8_t ttl; uint8_t protocol; uint16_t check; uint32_t saddr; uint32_t daddr; }; </pre>

A titolo di esempio si analizza l'intestazione di un pacchetto IPv4 relativo all'invio di un «ping», tra il nodo «A» e il nodo «B», della figura successiva.

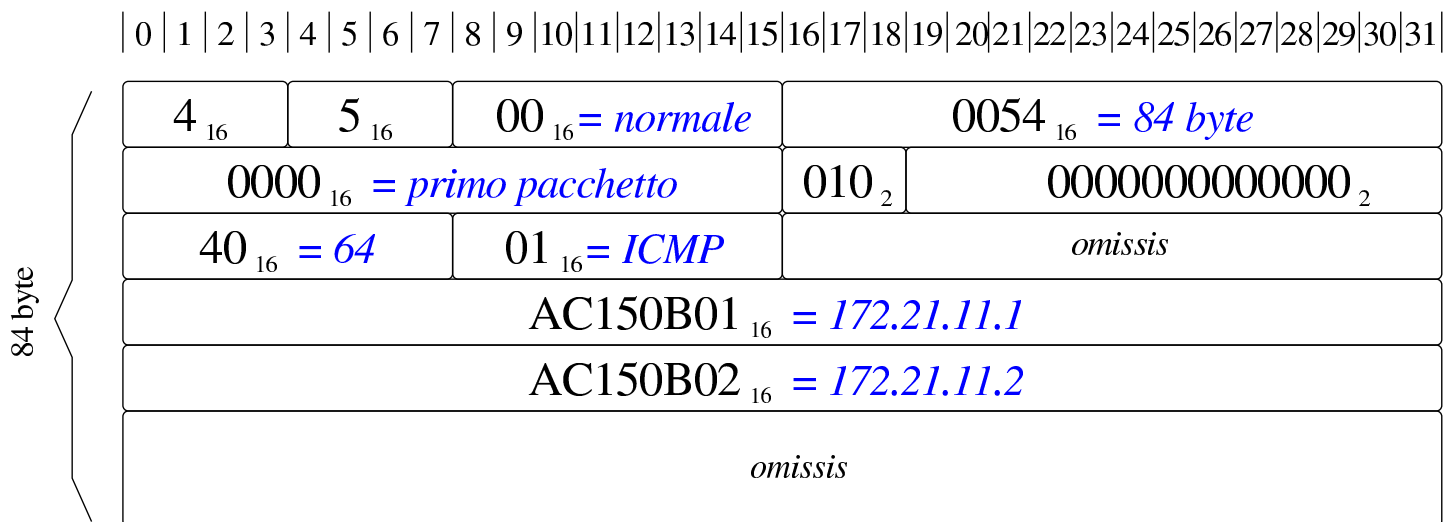
Figura 83.139. Situazione ipotetica di due nodi che utilizzano il protocollo IPv4.

IPv4: 172.21.11.1

IPv4: 172.21.11.2



Figura 83.140. Esempio di pacchetto «ping», inviato dall'indirizzo 172.21.11.1 a 172.21.11.2. Il pacchetto IPv4 non è frammentato ($MF == 0$) e non è nemmeno frammentabile ($DF == 1$).



Il codice di controllo che nell'esempio è stato omissis, viene calcolato sul contenuto dell'intestazione, considerando inizialmente che al posto del codice di controllo ci siano solo bit a zero. Il modo in cui questo viene calcolato è descritto nella sezione successiva; va tenuto conto, inoltre, che una volta calcolato questo viene collocato nell'intestazione invertendo i suoi bit (facendone il complemento a uno, ovvero applicando l'operatore binario NOT). Così facendo, per controllare la validità dell'intestazione, è sufficiente ripetere il calcolo del codice di controllo, utilizzando però questa volta anche quanto contenuto nel campo *header_checksum*, e verificando che il risultato sia pari a FFFF₁₆, oppure 0000₁₆.

È interessante osservare che, ogni volta che il campo *TTL* viene modificato da un router, questo deve provvedere ad aggiornare il codice di controllo dell'intestazione.

83.12.3 Il codice di controllo del TCP/IP

«

Il codice di controllo usato nell'intestazione dei pacchetti IPv4 e anche in altre situazioni, è calcolato suddividendo l'informazione di partenza in blocchi da 16 bit e sommando assieme questi blocchi, in modo binario, usando però l'aritmetica del complemento a uno.

Usando il sistema del complemento a uno, i numeri interi positivi si rappresentano in binario come di consueto, purché il bit più significativo sia pari a zero, mentre i numeri negativi sono rappresentati con il loro complemento a uno. Per esempio, disponendo di otto bit, il numero +5 si rappresenta come 00000101_2 , mentre il numero -5 diventa 11111010_2 . Pertanto, si distingue tra uno zero positivo (00000000_2) e uno zero negativo (11111111_2). Si osservino gli esempi seguenti:

+5 +	00000101 +	+5 +	00000101 +	+5 +	00000101 +
+2 =	00000010 =	-5 =	11111010 =	-7 =	11111000 =
----	-----	----	-----	----	-----
+7	00000111	0	11111111 (-0)	-2	11111101

Va però fatta attenzione ai riporti, perché questi vanno sommati al risultato:

+5 +	00000101 +	
-3 =	11111100 =	
----	-----	
+2	100000001	(si ottiene un riporto)
	00000001 +	
	1 =	(si somma il riporto)

	00000010	(questo è il risultato corretto)

Se si esegue una somma di più valori, i riporti si possono sommare tutti alla fine, senza farlo necessariamente a ogni coppia:

```

+7 +   00000111 +
-2 +   11111101 +
-3 +   11111100 =
-----
+2   1000000000 (si ottiene un riporto)

      00000000 +
          10 = (si somma il riporto)
-----
      00000010 (questo è il risultato corretto)

```

Per fare questo tipo di somma in un'architettura che utilizza l'aritmetica del complemento a due, è sufficiente utilizzare una variabile intera senza segno, di rango maggiore rispetto ai blocchi sommati, quindi si separano i riporti dal risultato per poi sommarli nuovamente a quello. Per esempio, nel caso del codice di controllo necessario ai protocolli TCP/IP, si utilizza una variabile intera, senza segno, a 32 bit. Si somma tutto quello che serve, quindi alla fine si separa il risultato contenuto nei 16 bit meno significativi, per sommarli i riporti contenuti nei 16 bit più significativi.

Nel listato successivo si vede come può essere realizzata una funzione per calcolare un codice di controllo relativo al contenuto di memoria che parte dalla posizione *data* e si estende per *size* byte. Nel procedimento va osservato il fatto che in memoria i dati si intendono essere ordinati nel modo naturale relativo alle comunicazioni di rete (*network byte order*); pertanto, nel calcolo viene usata la funzione *ntohs()* (*network to host short*) per garantire che i blocchi da 16 bit siano interpretati correttamente. Inoltre, dal momento che i

dati su cui calcolare il codice di controllo potrebbero essere composti da una quantità dispari di byte, l'ultimo otteetto viene trattato come se rappresentasse gli otto bit più significativi di un blocco di sedici.

Listato 83.144. Esempio di funzione per il calcolo della codice di controllo usato nei protocolli TCP/IP. La funzione *ntohs()* serve a garantire che i blocchi da 16 bit vengano interpretati nell'ordine giusto.

```
#include <stdint.h>
#include <arpa/inet.h>
uint16_t
checksum_tcpip (uint16_t *data, size_t size)
{
    int      i;
    uint32_t sum;
    uint16_t carry;
    uint16_t checksum;
    uint16_t last;
    uint8_t *octet;
    //
    // Somma
    //
    sum = 0;
    //
    for (i = 0; i < (size/2); i++)
        {
            sum += ntohs (data[i]);
        }
    //
    if (size % 2)
        {
            //
            // La dimensione è dispari, pertanto va considerato

```

```
    // anche l'ultimo ottetto.
    //
    octet = (uint8_t *) data;
    last = octet[size-1];
    last = last << 8;
    sum += last;
}
//
// Riporti
//
carry      = sum >> 16;
checksum   = sum & 0x0000FFFF;
checksum += carry;
//
//
//
return (checksum);
}
```

83.12.4 ICMP

Il protocollo ICMP si colloca al di sopra di quello IP, per l'invio di messaggi elementari, composti da un numero di messaggio (più precisamente si tratta di tipo e codice) con qualche informazione allegata. Il protocollo ICMP è molto importante per segnalare il fatto che un certo nodo non può essere raggiunto, ma spesso si usa per provare il funzionamento della rete con l'invio di una richiesta di eco, per la quale si attende una risposta equivalente.

Un pacchetto ICMP si inserisce all'interno di un pacchetto IP e si scompone come si vede nella figura successiva.

Figura 83.145. Struttura comune di un pacchetto del protocollo ICMP (all'interno di IPv4).

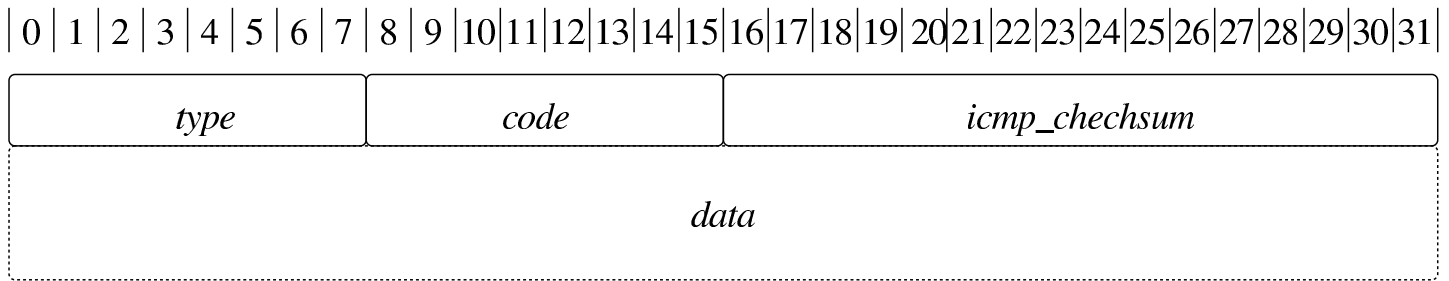


Tabella 83.146. Campi dell'intestazione comune di un pacchetto del protocollo ICMP.

Campo	Lunghezza	Descrizione
<i>type</i>	8 bit	Numero del tipo di messaggio: 01 ₁₆ <i>echo reply</i> 03 ₁₆ <i>destination unreachable</i> 08 ₁₆ <i>echo request</i>
<i>code</i>	8 bit	Qualificazione ulteriore del tipo di messaggio; di norma è pari a zero.
<i>icmp_checksum</i>	16 bit	Codice di controllo, calcolato sul pacchetto ICMP, a partire dal campo <i>type</i> , fino alla fine del pacchetto.
<i>data</i>	variabile	Contenuto del pacchetto ICMP che varia a seconda del tipo; in ogni caso, la dimensione massima dipende dalla dimensione massima di un pacchetto IPv4 non frammentato.

Tuttavia, il contenuto di un pacchetto ICMP può avere un'intestazione ulteriore, a seconda del tipo dichiarato nel campo

type.

Figura 83.147. Struttura di un pacchetto del protocollo ICMP, di tipo *echo request* o *echo reply*.

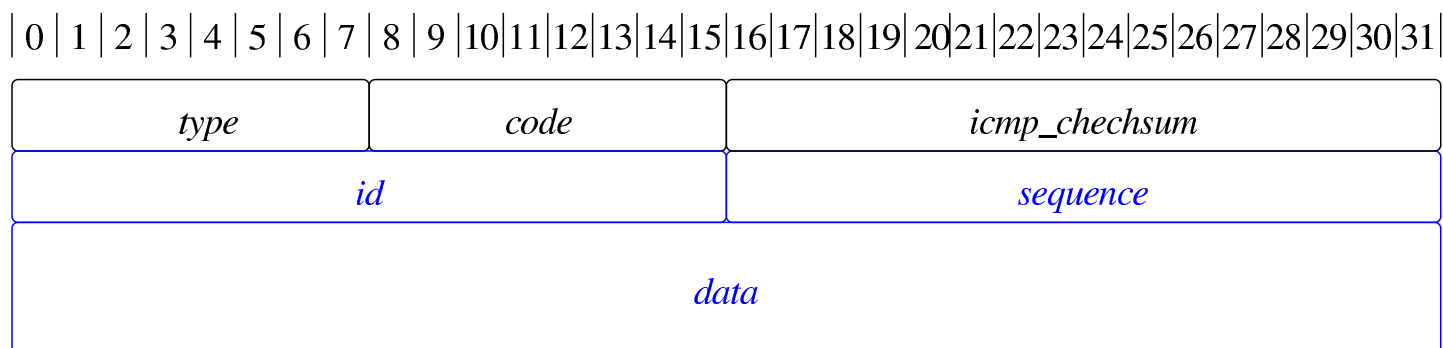


Tabella 83.148. Campi specifici dei pacchetti ICMP di eco.

Campo	Lunghezza	Descrizione
<i>id</i>	16 bit	Numero identificativo di una sequenza; nella richiesta di eco viene attribuito in modo casuale dal programma ' ping ' o da ciò che ne svolge la funzione.
<i>sequence</i>	16 bit	Numero di sequenza del pacchetto: la risposta a una richiesta di eco viene fatta usando lo stesso numero di identificazione e lo stesso numero di sequenza.

Campo	Lunghezza	Descrizione
<i>data</i>	variabile	Lo spazio rimanente nel pacchetto viene attribuito dal programma che richiede l'eco, in base alle esigenze; spesso si usano sequenze casuali di byte, per verificare il funzionamento della rete, dal momento che il pacchetto di risposta all'eco deve poi contenere gli stessi dati, confrontabili nell'origine.

Listato 83.149. Esempio di struttura in linguaggio C, per descrivere l'intestazione di un pacchetto del protocollo ICMP. Questa struttura vale indifferentemente per le architetture *little endian* e *big endian*.

```
struct icmphdr
{
    uint8_t  type;
    uint8_t  code;
    uint16_t checksum;
    union
    {
        struct
        {
            uint16_t id;
            uint16_t sequence;
        } __attribute__((packed)) echo;    // echo datagram
        uint32_t gateway;                  // gateway address
        struct
        {
            uint16_t unused;
            uint16_t mtu;
        } __attribute__((packed)) frag;    // path mtu discovery
    } un;
} __attribute__((packed));
```

A titolo di esempio si considerano due nodi, come nella figura successiva, e si analizza il contenuto di un pacchetto ICMP di richiesta di eco, da «A» a «B», seguito da una risposta conforme, in senso opposto.

Figura 83.150. Il nodo «A» invia una richiesta di eco al nodo «B» e il nodo «B» risponde.

IPv4: 172.21.11.15

IPv4: 172.21.254.254



Figura 83.151. Esempio di pacchetto ICMP, di tipo *echo request* (ping), inviato dall'indirizzo 172.21.11.15 a 172.21.254.254, completo dell'intestazione IPv4.

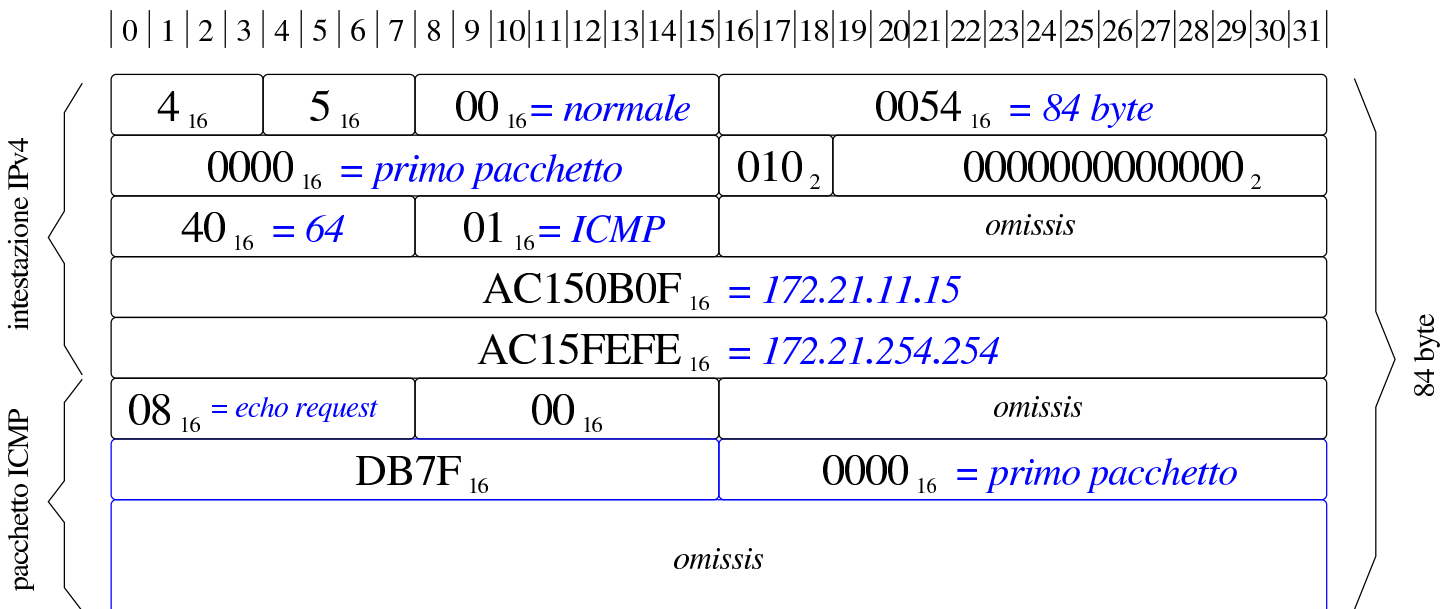
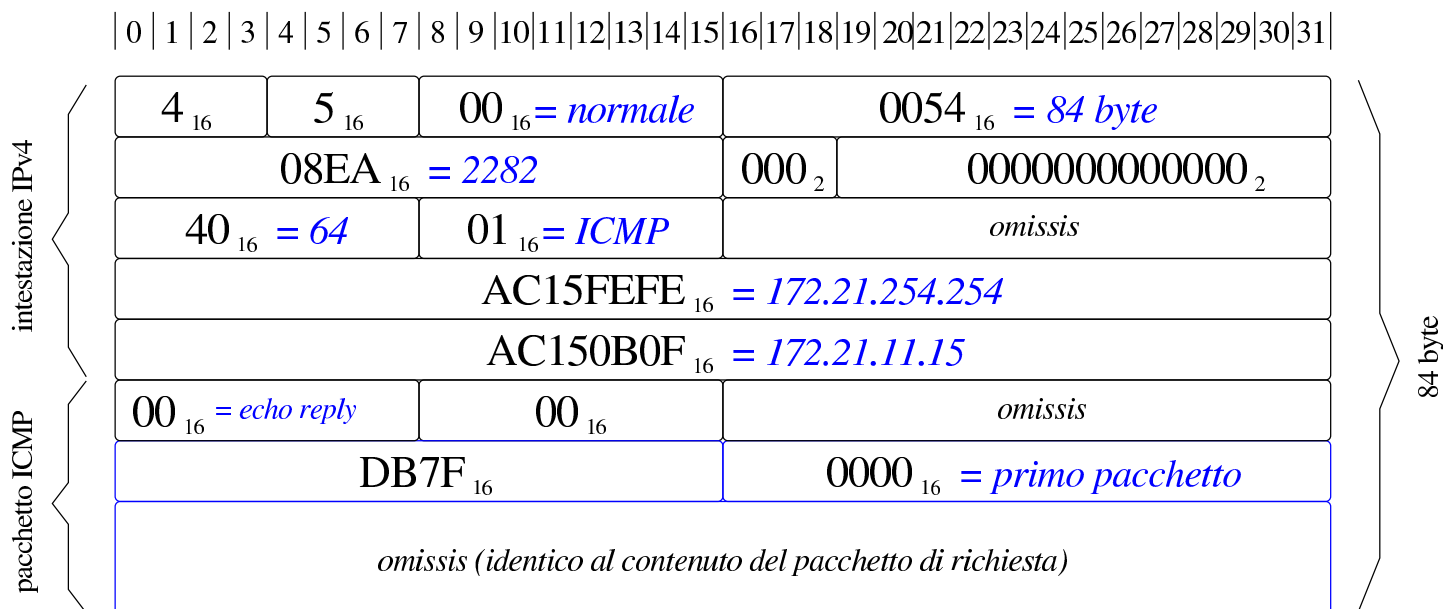


Figura 83.152. Esempio di pacchetto ICMP *echo reply* (pong), restituito da 172.21.254.254, completo di intestazione IPv4.



83.12.5 UDP

I pacchetti del protocollo UDP si inseriscono all'interno di pacchetti IP. I pacchetti UDP contengono a loro volta una propria intestazione, nella quale si prevede l'uso di un codice di controllo, relativo a tutto il pacchetto UDP, a cui però si aggiunge una pseudo-intestazione («pseudo», in quanto viene usata solo ai fini del calcolo del codice di controllo e non fa parte effettivamente del pacchetto). Il protocollo UDP inserisce il concetto di «porta» (*port*), distinto tra origine e destinazione.

Figura 83.153. Struttura effettiva di un pacchetto del protocollo UDP.

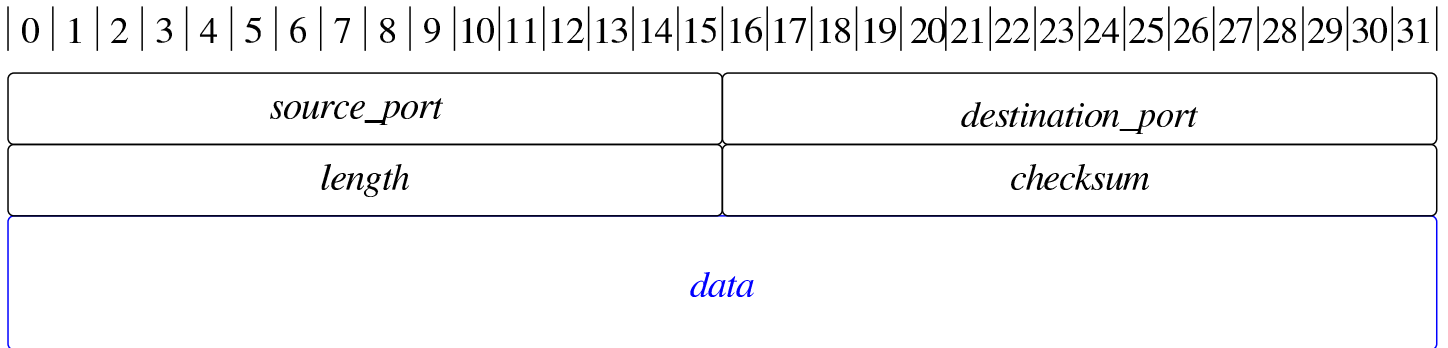


Figura 83.154. Pseudo-intestazione IPv4, utile per il calcolo del codice di controllo UDP, ma non facente parte del pacchetto.

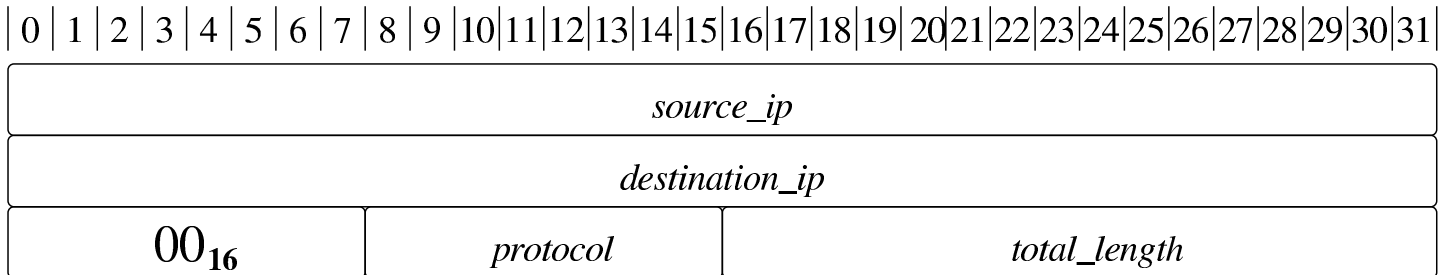


Tabella 83.155. Campi dell'intestazione UDP e della pseudo-intestazione relativa al protocollo IPv4.

Campo	Lunghezza	Descrizione
<i>source_port</i> <i>destination_port</i>	16 bit	Numero della porta di origine e di quella di destinazione del pacchetto.
<i>length</i>	16 bit	La lunghezza in ottetti (byte) del pacchetto UDP, composto da intestazione UDP e contenuto associato. Il valore minimo di questo valore è otto, ovvero il contenuto della sola intestazione.

Campo	Lunghezza	Descrizione
<i>checksum</i>	16 bit	Codice di controllo, ma facoltativo se usato nel protocollo IPv4. Per non applicare il codice di controllo, occorre mettere qui il valore zero. Se si vuole calcolare, questo deve riguardare tutto il pacchetto UDP e la pseudo-intestazione, con l'accuratezza di trasformare un eventuale risultato nullo nel valore $FFFF_{16}$.
<i>source_ip</i> <i>destination_ip</i>	32 bit	Come nell'intestazione IPv4.
<i>protocol</i>	16 bit	Il codice del protocollo, corrispondente in questo caso a 11_{16} , ovvero UDP.
<i>total_length</i>	16 bit	Lunghezza complessiva del pacchetto UDP, equivalente al campo <i>length</i> dell'intestazione UDP reale.

Listato 83.156. Esempio di struttura in linguaggio C, per descrivere l'intestazione di un pacchetto del protocollo UDP. Questa struttura vale indifferentemente per le architetture *little endian* e *big endian*.

```

struct udphdr
{
    uint16_t source;           // source port
    uint16_t dest;           // destination port
    uint16_t len;            // length
    uint16_t check;         // checksum
} __attribute__((packed));

```

A titolo di esempio si considerano due nodi, come nella figura successiva, e si analizza il contenuto di un pacchetto UDP, inviato da «A» a «B».

Figura 83.157. Dal nodo «A», porta 48281, parte un pacchetto UDP verso il nodo «B», alla porta 1234.

IPv4: 172.21.11.15:48281

IPv4: 172.21.254.254:1234



Figura 83.158. Esempio di pacchetto UDP, inviato dall'indirizzo 172.21.11.15, porta 48281, a 172.21.254.254 porta 1234, contenente la stringa 'ciao\n'. Il pacchetto è completo dell'intestazione IPv4 e i codici di controllo sono visibili.

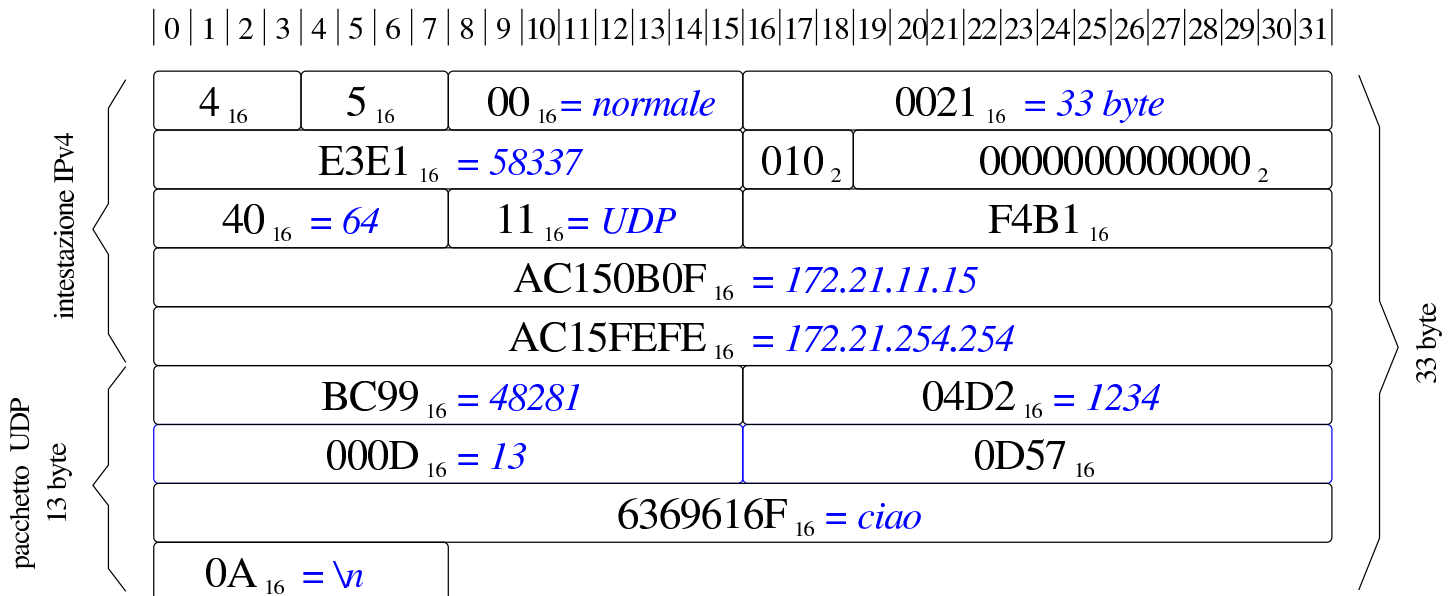


Figura 83.159. Pseudo-intestazione relativa al pacchetto di esempio della figura precedente.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$AC150B0F_{16} = 172.21.11.15$																															
$AC15FEFE_{16} = 172.21.254.254$																															
00_{16}				$11_{16} = 17$				$000D_{16} = 13$																							

Nel listato successivo si vede un piccolo programma che calcola il codice di controllo del pacchetto IPv4 dell'esempio.

Listato 83.160. Calcolo del codice di controllo nell'intestazione IPv4 dell'esempio. Una volta compilato, il programma visualizza correttamente il valore atteso: $F4B1_{16}$.

```
#include <stdint.h>
#include <stdio.h>
int
main (void)
{
    uint32_t sum = 0;
    uint16_t carry;
    uint16_t checksum;
    //
    sum += 0x4500;
    sum += 0x0021;
    sum += 0xe3e1;
    sum += 0x4000;
    sum += 0x4011;
    sum += 0x0000;
    sum += 0xac15;
    sum += 0x0b0f;
    sum += 0xac15;
    sum += 0xfefe;
    //
```

```
carry      = sum >> 16;
checksum   = sum & 0x0000FFFF;
checksum += carry;
//
checksum = ~checksum;
//
printf ("0x%04x\n", checksum);
//
return 0;
}
```

Listato 83.161. Calcolo del codice di controllo nell'intestazione UDP dell'esempio che tiene conto della pseudo-intestazione. Una volta compilato, il programma visualizza correttamente il valore atteso: $0D57_{16}$.

```
#include <stdint.h>
#include <stdio.h>
int
main (void)
{
    uint32_t sum = 0;
    uint16_t carry;
    uint16_t checksum;
    //
    sum += 0xbc99;
    sum += 0x04d2;
    sum += 0x000d;
    sum += 0x0000;
    sum += 0x6369;
    sum += 0x616f;
    sum += 0x0a00;

    sum += 0xac15;
    sum += 0x0b0f;
```

```
sum += 0xac15;
sum += 0xfefe;
sum += 0x0011;
sum += 0x000d;
//
carry      = sum >> 16;
checksum   = sum & 0x0000FFFF;
checksum += carry;
//
checksum = ~checksum;
//
if (checksum == 0)
    {
        checksum = 0xffff;
    }
//
printf ("0x%04x\n", checksum);
//
return 0;
}
```

83.12.6 TCP

Il protocollo TCP si distingue da UDP in quanto permette di stabilire un flusso di dati bidirezionale tra due porte di due nodi. Il processo che inizia una connessione TCP, apre una porta presso il nodo locale in cui si trova a funzionare, contattando una porta di un altro nodo, presso la quale si deve trovare un altro processo in attesa. Successivamente i processi coinvolti non si preoccupano di altro, a parte il fatto di trasmettere e ricevere dati attraverso il canale costituito dalla connessione. Infatti, la gestione della connessione TCP avviene per opera del sistema operativo, attraverso l'invio e la ricezione dei pac- <<

chetti relativi, con tutti i controlli necessari a garantire la correttezza del flusso di dati.

L'intestazione di un pacchetto del protocollo TCP contiene degli indicatori (*flag*), alcuni dei quali sono essenziali e appaiono descritti nella tabella successiva:

Indicatore	Denominazione	Descrizione
ACK	<i>acknowledge</i>	Si conferma la ricezione, specificando qual è il prossimo byte che si attende di ricevere dalla controparte.
PSH	<i>push</i>	Si richiede che i dati trasmessi fino a quel punto siano recapitati senza indugio al processo in ricezione dall'altra parte.
RST	<i>reset</i>	Azzeramento della connessione.
SYN	<i>synchronization</i>	Inizio di una connessione.
FIN	<i>finalization</i>	Conclusione della trasmissione.

Il protocollo TCP, gestito dal sistema operativo, richiede che la ricezione dei pacchetti contenenti dati sia confermata dalla controparte. Ma non è strettamente necessario confermare ogni pacchetto ricevuto, in quanto il riferimento è al byte *n*-esimo, che quindi convalida anche quelli ricevuti in precedenza. In tal modo, una delle due parti può tentare di trasmettere più pacchetti in rapida successione, prima di ricevere una conferma; poi, se la conferma arriva solo parzialmente, può ritrasmettere a partire dalla porzione non ancora confermata.

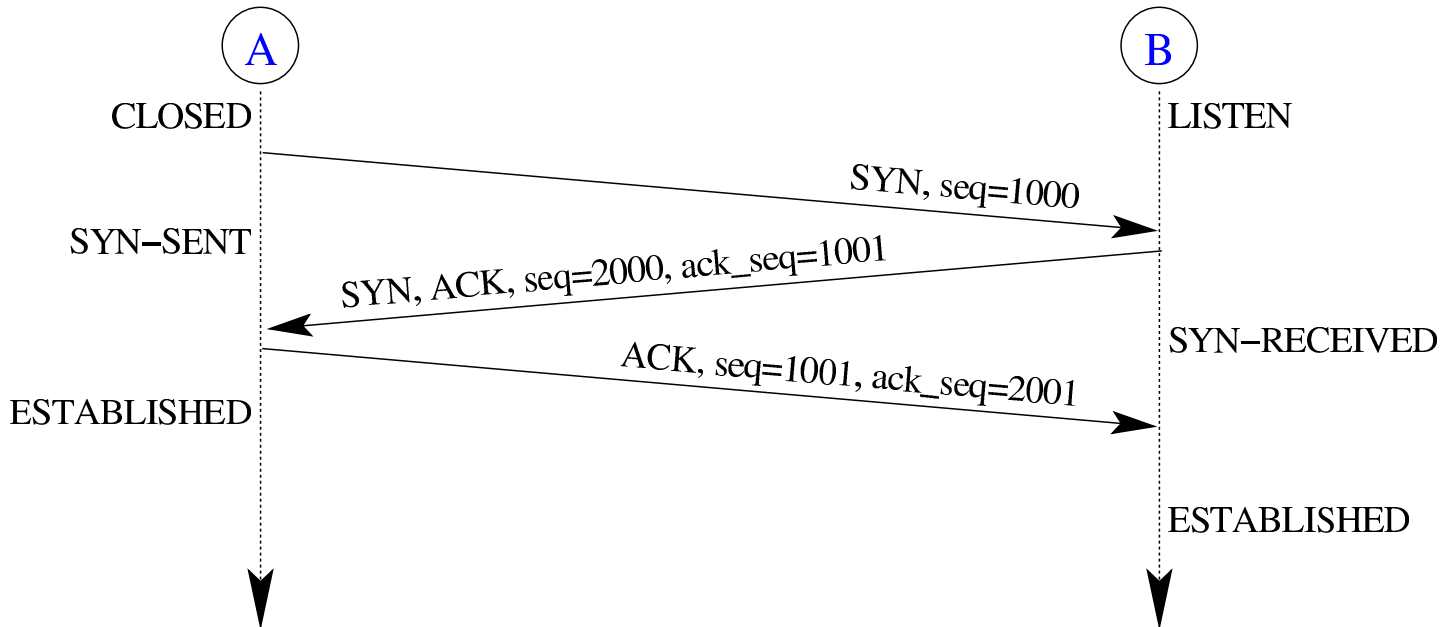
Una connessione TCP prevede undici stati, descritti dalla tabella successiva:

Stato	Descrizione
LISTEN	Si è in attesa di una richiesta di connessione. Questa condizione si verifica quando un processo apre una porta TCP locale e attende di ricevere una richiesta da un altro processo (locale o remoto) per poter instaurare con lui una connessione.
SYN-SENT	È già stato inviato un pacchetto SYN e si è in attesa di un pacchetto SYN di risposta. Si verifica questa condizione quando un processo tenta di contattarne un altro, già in ascolto su una certa porta di un certo nodo.
SYN-RECEIVED	È stato ricevuto un pacchetto SYN e a questo è stato risposto con una conferma e un altro pacchetto SYN, del quale si attende conferma (ACK). Dopo la conferma attesa, si passa allo stato successivo: ESTABLISHED.
ESTABLISHED	La connessione è stata instaurata e il flusso dei dati, nelle due direzioni, può avere luogo.
FIN-WAIT-1	L'applicazione locale (rispetto alla connessione) ha chiuso il proprio flusso di trasmissione ed è stato inviato un pacchetto FIN alla controparte, rimanendo in attesa di una conferma o di un altro pacchetto FIN.
CLOSE-WAIT	È stato ricevuto un pacchetto FIN dalla controparte ed è stata trasmessa la conferma relativa, mentre l'applicazione locale non ha ancora chiuso il proprio flusso in uscita (in scrittura).
FIN-WAIT-2	Dopo lo stato FIN-WAIT-1 è stata ricevuta la conferma e si passa quindi all'attesa che la controparte completi l'invio di dati con un pacchetto FIN.
CLOSING	Dopo lo stato FIN-WAIT-1, prima di ricevere una conferma dall'altra parte, è stato ricevuto un pacchetto FIN ed è stata inviata conferma di questo: si attende quindi la conferma al proprio pacchetto FIN.

Stato	Descrizione
LAST-ACK	È stato ricevuto un pacchetto FIN, è stata inviata conferma e successivamente è stato inviato un pacchetto FIN: si rimane quindi in attesa di una conferma dall'altra parte.
TIME-WAIT	I due lati della connessione sono stati chiusi con i pacchetti FIN e le conferme rispettive sono state inviate: si attende comunque qualche tempo prima di eliminare la connessione dalla gestione del sistema. Il tempo previsto è di 2 MLS (praticamente 4 minuti).
CLOSED	Condizione immaginaria di una connessione ormai chiusa e dimenticata dal sistema di gestione del protocollo TCP.

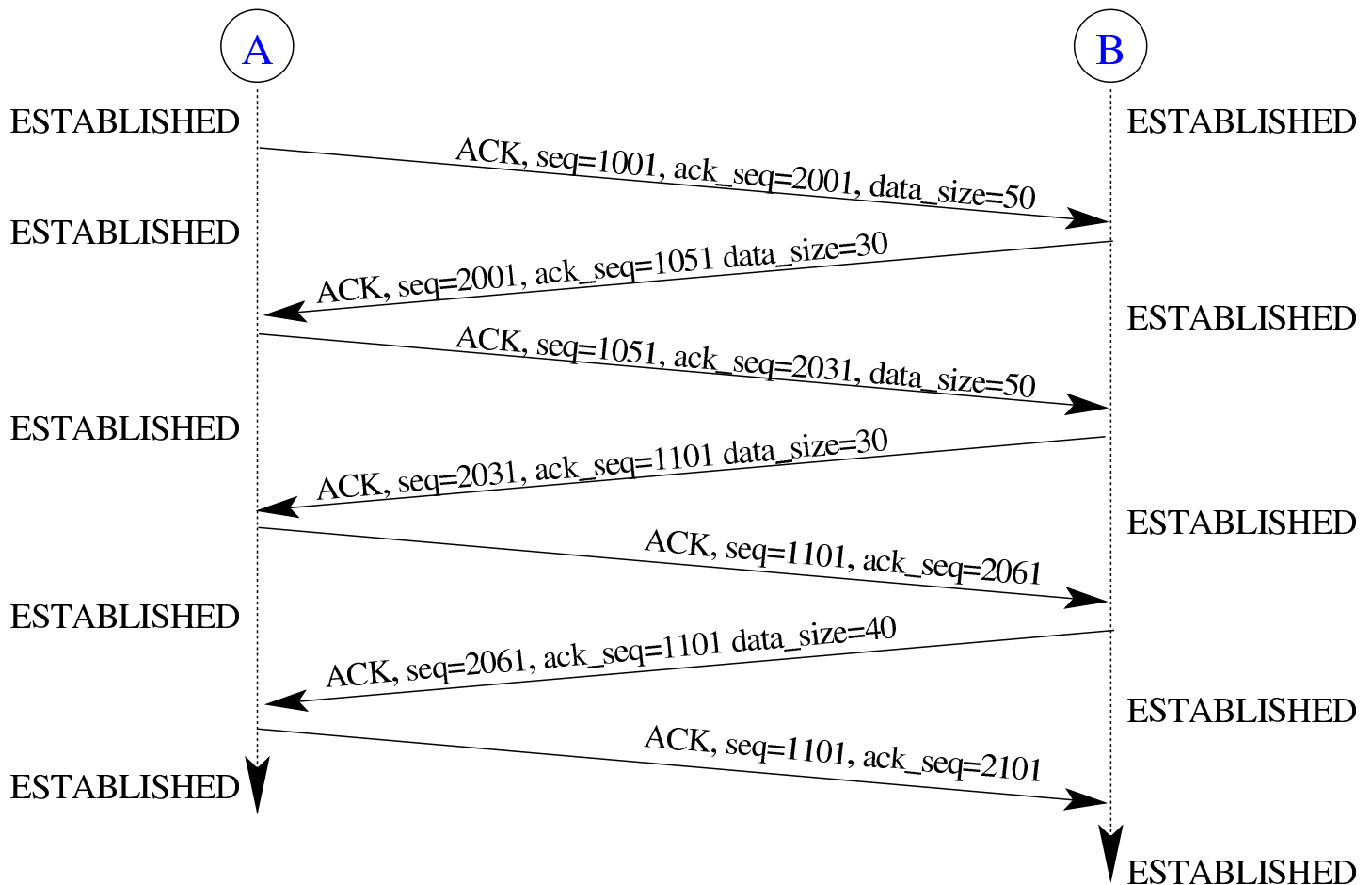
Nelle figure successive si esemplifica il procedere degli stati di una connessione, partendo dalla sua creazione, fino alla sua conclusione, ipotizzando un breve scambio di dati. Ognuno dei lati della connessione decide qual è il proprio numero iniziale di sequenza; da quel punto in poi, l'incremento di quel valore serve a consentire la verifica dell'ordine che devono avere i pacchetti. Il valore rappresentato dalla metavariable *seq* è il numero di sequenza iniziale del pacchetto, mentre *ack_seq* è il valore di sequenza che ci si attende di ricevere. Per esempio, un valore di *ack_seq* pari a 1234 significa che sono stati ricevuti dati fino al byte corrispondente alla sequenza 1233 e, se altri dati devono giungere, il prossimo byte da ricevere deve essere quello con il numero di sequenza 1234. In ogni caso, va osservato che nella creazione della connessione e nella sua conclusione, c'è un momento in cui il numero di sequenza viene incrementato di una unità, senza che ciò sia dovuto alla trasmissione effettiva di un byte.

Figura 83.164. Negoziazione iniziale: *three way handshake*. Dal lato «A» si inizia il procedimento per instaurare una connessione con il lato «B». La sequenza iniziale dal lato «A» è pari a 1000, mentre quella iniziale dal lato «B» è qui pari a 2000.



Dopo la negoziazione con i valori ipotizzati nell'esempio, il primo byte che «A» può trasmettere a «B» ha il numero di sequenza 1001, mentre nel senso opposto questo numero è 2001. Nella figura successiva, «A» e «B» si inviano dati reciprocamente per 100 byte ciascuno.

Figura 83.165. Quando i due lati della connessione sono pronti, le due parti possono trasmettersi dei dati. In questo caso si ipotizza che ogni pacchetto sia sempre confermato.



Dopo il breve scambio di dati, il lato «A» decide di chiudere la propria trasmissione (scrittura), informando la controparte, la quale, tuttavia, rimane nella facoltà di continuare a inviare dati.

Figura 83.166. Il lato «A» chiude il suo canale di trasmissione.

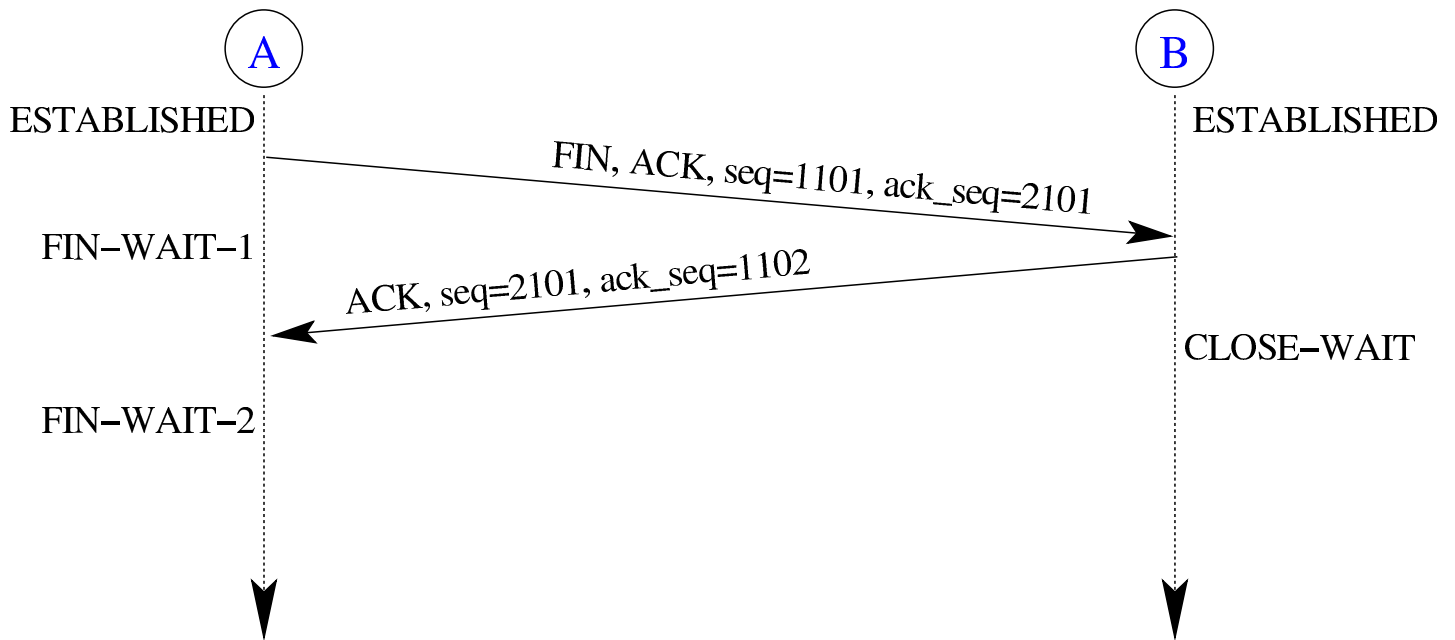


Figura 83.167. Dopo la chiusura dal lato «A», si ipotizza che il lato «B» continui a trasmettere in tutto altri 100 byte di dati.

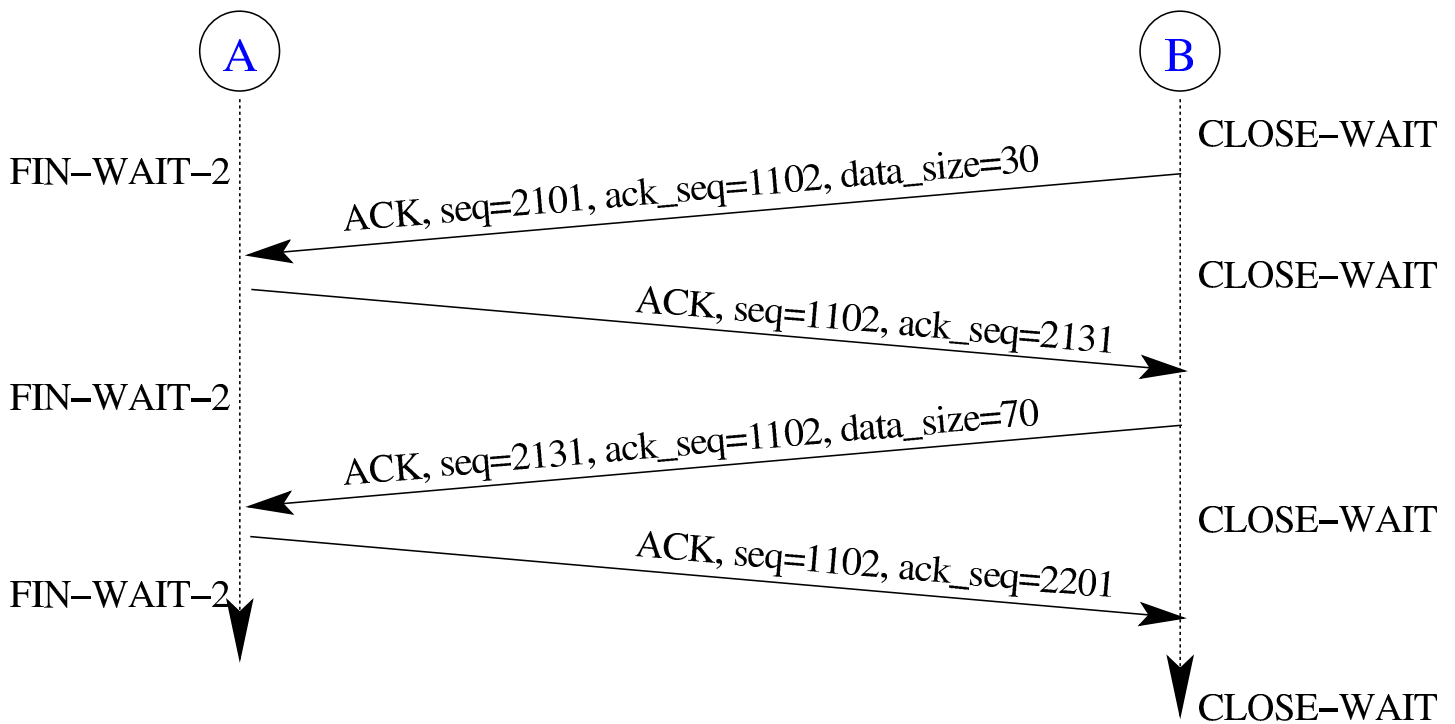
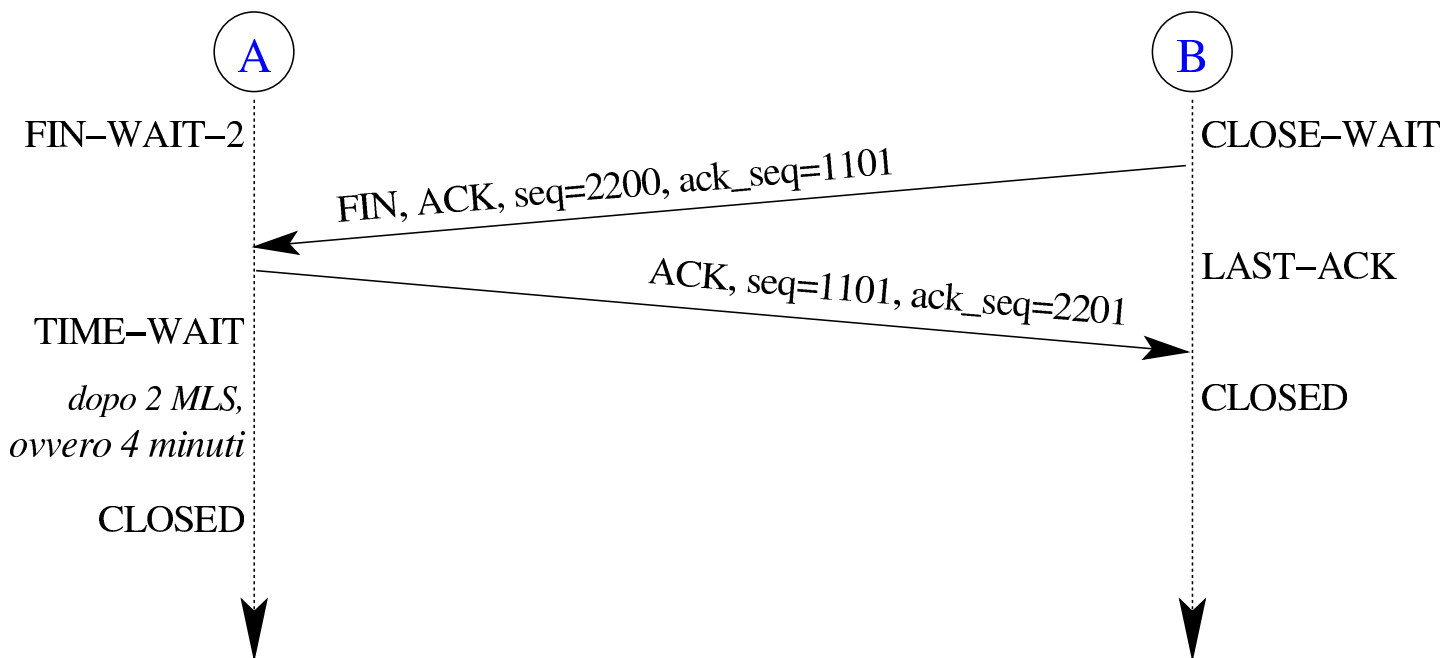
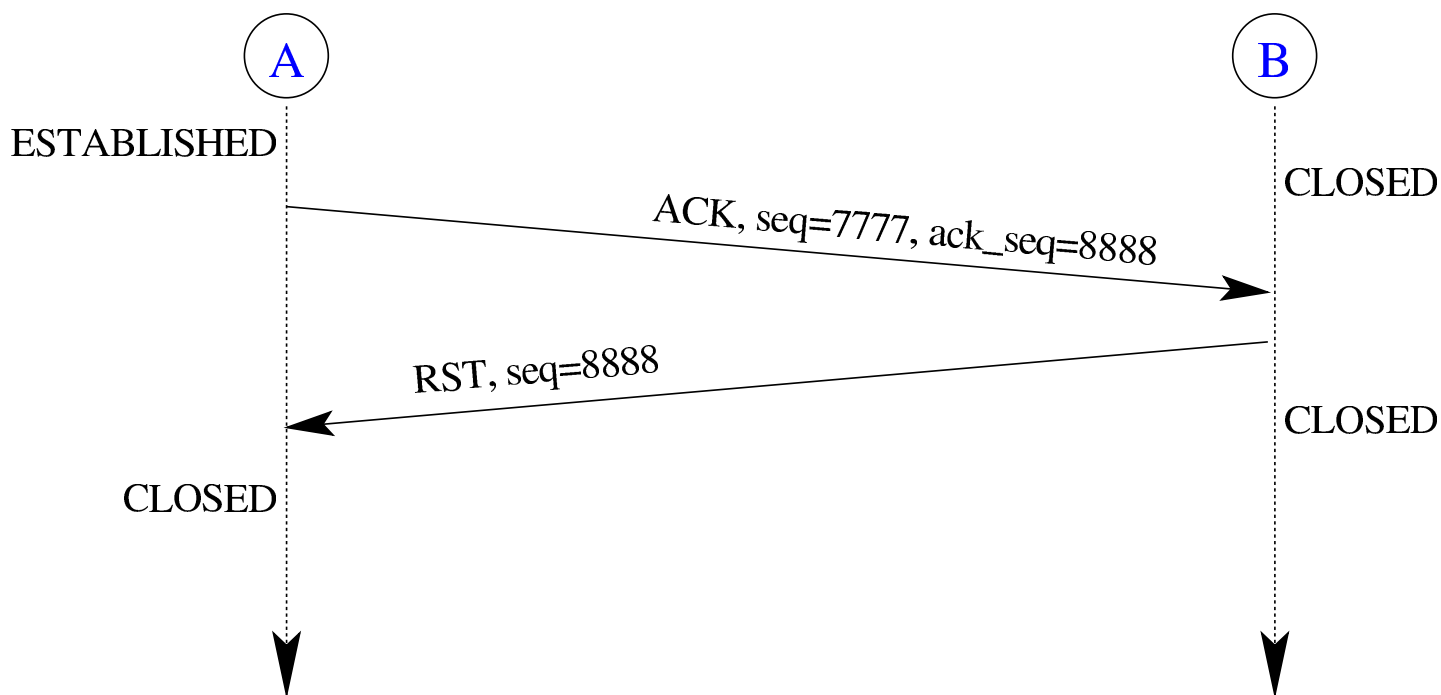


Figura 83.168. Dopo la chiusura dal lato «A», e dopo che il lato «B» ha finito con la propria trasmissione, anche questo decide di chiudere e si arriva alla conclusione della connessione. Tuttavia, il lato «A» che alla fine non può sapere se il lato «B» ha ricevuto effettivamente la conferma, rimane nello stato di TIME-WAIT per un certo ammontare di tempo, prima che per lui la connessione si possa considerare completamente chiusa.



Quando una delle parti viene confusa per qualche motivo, ricevendo un pacchetto che non sa qualificare nella connessione in corso o perché non fa proprio parte di una connessione, la risposta avviene attraverso un pacchetto con l'indicatore RST attivo.

Figura 83.169. Il ricevimento di un pacchetto fuori contesto, provoca una risposta di azzeramento.



Nella figura successiva si vede la struttura effettiva di un pacchetto TCP, da considerare all'interno di un pacchetto IPv4. Le opzioni sono facoltative e non sempre vengono trasportati dei dati.

Figura 83.170. Struttura effettiva di un pacchetto del protocollo TCP.

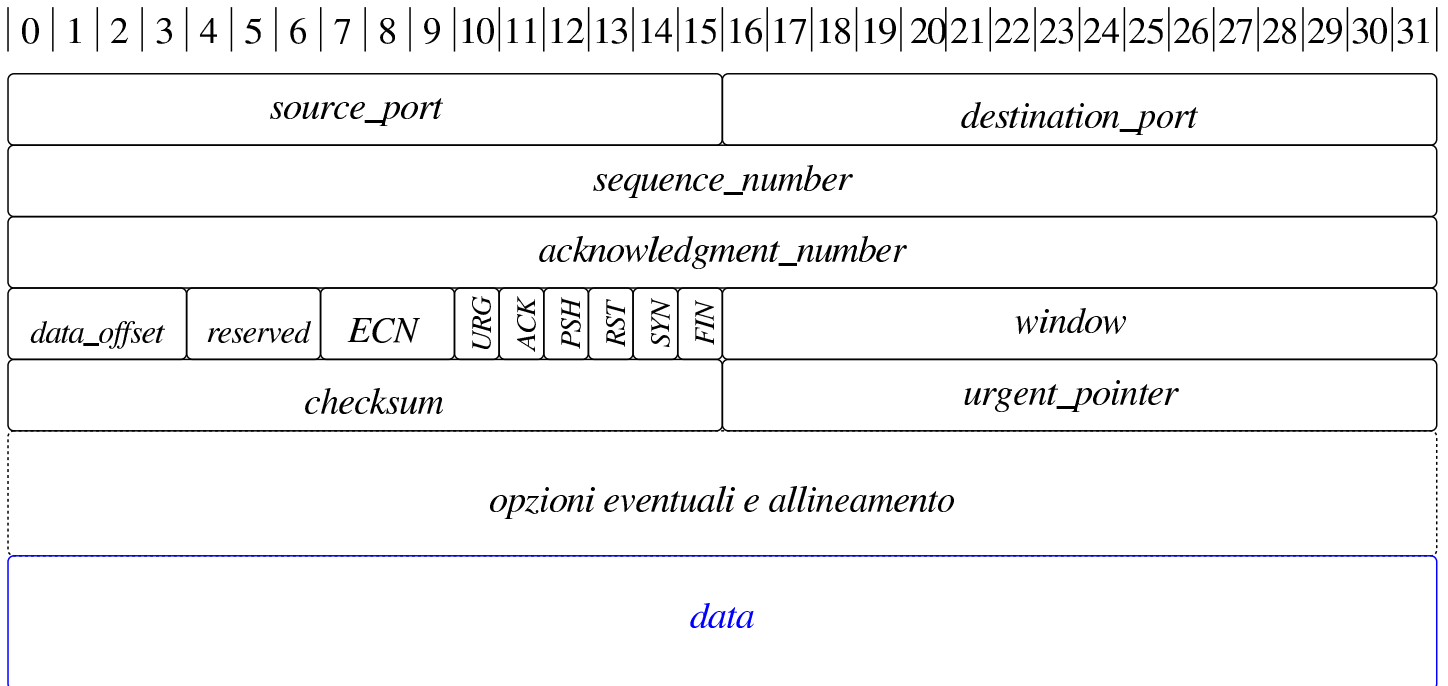


Figura 83.171. Pseudo-intestazione IPv4, utile per il calcolo del codice di controllo TCP, ma non facente parte del pacchetto.

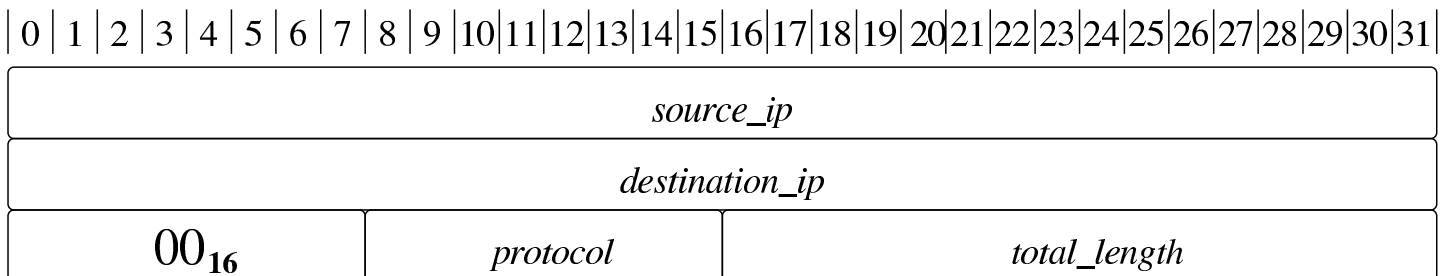


Tabella 83.172. Campi dell'intestazione TCP e della pseudo-intestazione relativa al protocollo IPv4.

Campo	Lunghezza	Descrizione
<i>source_port</i> <i>destination_port</i>	16 bit	Numero della porta di origine e di quella di destinazione del pacchetto.

Campo	Lunghezza	Descrizione
<i>sequence_number</i>	32 bit	Numero di sequenza del primo byte di dati allegati, ammesso che ci siano effettivamente dati contenuti nel pacchetto. Va osservato che quando viene inviato un pacchetto SYN, questo numero rappresenta la sequenza iniziale della connessione, mentre il primo byte di dati che può poi essere trasmesso deve avere un numero di sequenza pari a questo valore più una unità.
<i>acknowledgment_number</i>	32 bit	Numero di sequenza del primo byte di dati che si attende di ricevere dalla controparte, in quanto le sequenze precedenti sono già state ricevute correttamente.
<i>data_offset</i>	4 bit	La quantità di blocchi a 32 bit di cui si compone l'intestazione, opzioni incluse; in pratica, moltiplicando questo valore per quattro, si ottiene il puntatore al primo byte di dati contenuto nel pacchetto.
<i>reserved</i>	3 bit	Un campo da lasciare a zero.
<i>ECN</i>	3 bit	<i>Explicit congestion notification</i> Il contenuto di questo campo è definito dal RFC 3160. In condizioni normali, questo campo viene lasciato a zero.

Campo	Lunghezza	Descrizione
URG	1 bit	<i>urgent</i> Se l'indicatore è attivo (ha il valore 1), significa che il valore contenuto nel campo <i>urgent pointer</i> è significativo. In condizioni normali, questo indicatore è a zero e il campo <i>urgent_pointer</i> è a zero.
ACK	1 bit	<i>acknowledge</i> Se l'indicatore è attivo significa che il contenuto del campo <i>acknowledgement_sequence</i> indica il numero di sequenza del primo byte di dati attesi dalla controparte; diversamente, significa che quel numero di sequenza non è da considerare.
PSH	1 bit	<i>push</i> Se l'indicatore è attivo significa che si sta chiedendo di recapitare senza indugio i dati trasmessi fino a questo punto, all'applicazione di destinazione.

Campo	Lunghezza	Descrizione
<i>RST</i>	1 bit	<i>reset</i> Se l'indicatore è attivo significa che è stato ricevuto un pacchetto TCP dalla controparte con un numero <i>acknowledgement_sequence</i> pari al numero di sequenza del pacchetto attuale, ma ciò risulta al di fuori di una connessione conosciuta e si richiede pertanto la cancellazione di tale comunicazione.
<i>SYN</i>	1 bit	<i>synchronization</i> Se l'indicatore è attivo significa che si sta tentando di instaurare una connessione.
<i>FIN</i>	1 bit	<i>finalization</i> Se l'indicatore è attivo significa che si sta chiudendo il canale di trasmissione verso la controparte.
<i>Window</i>	16 bit	Si tratta della «finestra di ricezione», pari alla quantità di byte che possono essere ricevuti simultaneamente, in uno o più pacchetti successivi.
<i>checksum</i>	16 bit	Codice di controllo, relativo a tutto il pacchetto TCP e alla pseudo-intestazione (come per UDP).

Campo	Lunghezza	Descrizione
<i>urgent_pointer</i>	16 bit	Puntatore a dati «urgenti», valido solo se l'indicatore URG risulta attivo.
<i>source_ip</i> <i>destination_ip</i>	32 bit	Come nell'intestazione IPv4.
<i>protocol</i>	16 bit	Il codice del protocollo, corrispondente in questo caso a 06 ₁₆ , ovvero TCP.
<i>total_length</i>	16 bit	Lunghezza complessiva del pacchetto TCP.

Listato 83.173. Esempio di struttura in linguaggio C, per descrivere l'intestazione di un pacchetto del protocollo TCP. A sinistra nella versione *little endian*; a destra in quella *big endian*. I campi *res1* e *res2* includono i tre bit riservati successivi a *data_offset* e il campo *ECN*.

<i>little endian</i>	<i>big endian</i>
<pre> struct tcphdr { uint16_t source; uint16_t dest; uint32_t seq; uint32_t ack_seq; uint16_t res1 : 4, doff : 4, fin : 1, syn : 1, rst : 1, psh : 1, ack : 1, urg : 1, res2 : 2; uint16_t window; uint16_t check; uint16_t urg_ptr; }; </pre>	<pre> struct tcphdr { uint16_t source; uint16_t dest; uint32_t seq; uint32_t ack_seq; uint16_t doff : 4, res1 : 4, res2 : 2, urg : 1, ack : 1, psh : 1, rst : 1, syn : 1, fin : 1; uint16_t window; uint16_t check; uint16_t urg_ptr; }; </pre>

A titolo di esempio si considerano due nodi, come nella figura successiva, e si analizza il contenuto di un pacchetto TCP, inviato da «A» a «B», nell'ambito di una connessione in corso (entrambi i lati sono nella condizione CONNECTED).

Figura 83.174. Dal nodo «A», porta 11111, parte un pacchetto TCP verso il nodo «B», alla porta 12345.

IPv4: 172.21.11.15:11111

IPv4: 172.21.254.254:12345



Figura 83.175. Esempio di pacchetto TCP, inviato dall'indirizzo 172.21.11.15, porta 11111, a 172.21.254.254 porta 12345, contenente la stringa 'ciao\n'. Il pacchetto è completo dell'intestazione IPv4 e i codici di controllo sono visibili.

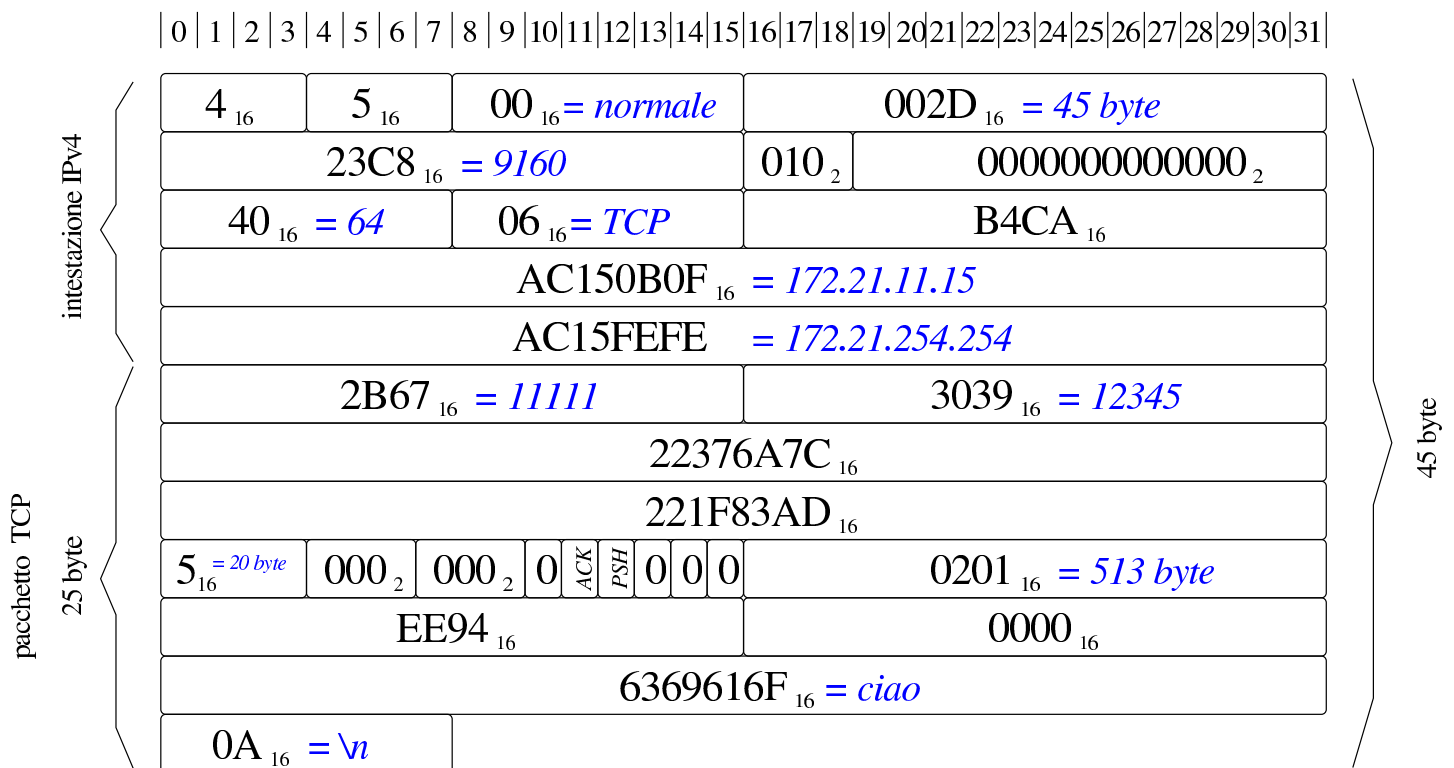


Figura 83.176. Pseudo-intestazione relativa al pacchetto di esempio della figura precedente.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$AC150B0F_{16} = 172.21.11.15$																															
$AC15FEFE_{16} = 172.21.254.254$																															
00_{16}				$06_{16} = TCP$								$0019_{16} = 25$																			

Nel listato successivo si vede un piccolo programma che calcola il codice di controllo del pacchetto IPv4 dell'esempio.

Listato 83.177. Calcolo del codice di controllo nell'intestazione IPv4 dell'esempio. Una volta compilato, il programma visualizza correttamente il valore atteso: $B4CA_{16}$.

```
#include <stdint.h>
#include <stdio.h>
int
main (void)
{
    uint32_t sum = 0;
    uint16_t carry;
    uint16_t checksum;
    //
    sum += 0x4500;
    sum += 0x002D;
    sum += 0x23C8;
    sum += 0x4000;
    sum += 0x4006;
    sum += 0x0000;
    sum += 0xAC15;
    sum += 0x0B0F;
    sum += 0xAC15;
    sum += 0xFEFE;
    //
```

```
carry      = sum >> 16;
checksum   = sum & 0x0000FFFF;
checksum += carry;
//
checksum = ~checksum;
//
printf ("0x%04x\n", checksum);
//
return 0;
}
```

Listato 83.178. Calcolo del codice di controllo nell'intestazione TCP dell'esempio che tiene conto della pseudo-intestazione. Una volta compilato, il programma visualizza il valore: EE94₁₆.

```
#include <stdint.h>
#include <stdio.h>
int
main (void)
{
    uint32_t sum = 0;
    uint16_t carry;
    uint16_t checksum;
    //
    sum += 0x2B67;
    sum += 0x3039;
    sum += 0x2237;
    sum += 0x6A7C;
    sum += 0x221F;
    sum += 0x83AD;
    sum += 0x5018;
    sum += 0x0201;
    sum += 0x0000;
    sum += 0x0000;
    sum += 0x6369;
```

```
sum += 0x616F;
sum += 0x0A00;

sum += 0xac15;
sum += 0x0b0f;
sum += 0xac15;
sum += 0xfefe;
sum += 0x0006;
sum += 0x0019;
//
carry      = sum >> 16;
checksum   = sum & 0x0000FFFF;
checksum += carry;
//
checksum = ~checksum;
//
printf ("0x%04x\n", checksum);
//
return 0;
}
```

83.13 Riferimenti



- *Intel® 64 and IA-32 Architectures Software Developer's Manuals*, <http://developer.intel.com/products/processor/manuals/index.htm>
- David Brackeen, *256-Color VGA Programming in C*, <http://www.brackeen.com/vga/>
- Brandon Friesen, *Bran's kernel development tutorial*, <http://www.osdever.net/bkerndev/Docs/title.htm>

- Wikipedia, *Global Descriptor Table*, http://en.wikipedia.org/wiki/Global_Descriptor_Table
- *Higher Half With GDT*, http://wiki.osdev.org/Higher_Half_With_GDT
- Weqaar A. Janjua, *IA-32 Boot sector code*, <http://sites.google.com/site/weqaar/Home/files>
- Gergor Brunmar, *The world of Protected mode*, http://www.osdever.net/tutorials/pdf/gb_pmode.pdf
- John Fine, *Descriptor Tables: GDT, IDT and LDT*, <http://www.osdever.net/tutorials/pdf/descriptors.pdf>, <http://www.osdever.net/tutorials/view/descriptor-tables-gdt-idt-ldt>
- Jochen Liedtke, *Segments, Intel's IA-32 from a system architecture view*, <http://wayback.archive.org/web/2004/http://i30ww30w.ira.uka.de/teaching/coursedocuments/48/segments.pdf>
- Allan Cruse, *CS 630: Advanced Microcomputer Programming*, <http://www.cs.usfca.edu/~cruse/cs630f06/>
- Wikipedia, *Interrupt descriptor table*, http://en.wikipedia.org/wiki/Interrupt_descriptor_table
- *Interrupt Descriptor Table*, http://wiki.osdev.org/Interrupt_Descriptor_Table
- Alexander Blessing, *Programming the PIC*, <http://www.osdever.net/tutorials/pdf/pic.pdf>
- *8259 Programmable Interrupt Controller (PIC)*, <http://www.pklab.net/index.php?id=94>
- *Write your own Operating System: Interrupt Service Routines*, http://wiki.osdev.org/Interrupt_Service_Routines

- Wikipedia, *Intel 8253*, http://en.wikipedia.org/wiki/Intel_8253
- *Write your own Operating System: Programmable Interval Timer*, http://wiki.osdev.org/Programmable_Interval_Timer
- Mark Feldman, *Programming the Intel 8253 Programmable Interval Timer*, <http://www.nondot.org/sabre/os/files/MiscHW/PIT.txt>
- Salvatore D'Angelo, *Keyboard Driver*, <http://opencommunity.altervista.org/samples/openjournal/keyboard.html>
- Adam Chapweske, *The PS/2 Keyboard Interface*, http://www.burtonsys.com/ps2_chapweske.htm , <http://www.tayloredge.com/reference/Interface/atkeyboard.pdf>
- OSDev Wiki, *ATA PIO mode*, http://wiki.osdev.org/ATA_PIO_Mode
- T13, *AT Attachment with Packet Interface - 6 (ATA/ATAPI-6)*, <http://bos.asmhackers.net/docs/ata/docs/ata-atapi-6-3b.pdf>
- LDP, *PCI*, <http://tldp.org/LDP/tlk/dd/pci.html>
- OSDev Wiki, *PCI*, <http://wiki.osdev.org/PCI>
- PLX technology, *pci 9054*, <http://www.nikhef.nl/~peterj/datasheets/9054db54-1C.pdf>
- *PCI and AGP Vendors, Devices and Subsystems identification file*, http://wayback.archive.org/web/2009*/http://members.datafast.net.au/~dft0802/ , http://wayback.archive.org/web/2009*/http://members.datafast.net.au/~dft0802/downloads.htm
- OSDev Wiki, *NE2000*, <http://wiki.osdev.org/Ne2000>

- National semiconductor, *DP8390D/NS32490D NIC network interface controller*, <http://www.national.com/pf/DP/DP8390D.html>
- National semiconductor, *Writing drivers for DP8390 NIC family of ethernet controllers*, <http://www.datasheetarchive.com/Indexer/Datasheet-017/DSA00296168.html>
- Realtek, *RTL8019AS, Realtek Full-Duplex Ethernet Controller with Plug and Play Function (RealPNP), SPECIFICATION*, <http://www.ethernut.de/pdf/8019as19ds.pdf>
- Network Working Group, *RFC 826: An Ethernet Address Resolution Protocol*, 1982, <http://www.ietf.org/rfc/rfc826.txt>
- Information Sciences Institute, University of Southern California, *RFC 791: Internet protocol*, 1981, <http://www.ietf.org/rfc/rfc791.txt>
- J. Postel, *RFC 792: Internet control message protocol*, 1981, <http://www.ietf.org/rfc/rfc792.txt>
J. Mogul, J. Postel, *RFC 950: Internet standard subnetting procedure*, 1985, <http://www.ietf.org/rfc/rfc950.txt>
- J. Postel, *RFC 768: User datagram protocol*, 1980, <http://www.ietf.org/rfc/rfc768.txt>
- Information science institute, *RFC 793: Transmission control protocol*, 1981, <http://www.ietf.org/rfc/rfc793.txt>

¹ La modalità protetta è quella che consente di accedere alla memoria oltre il limite di 1 Mibyte.

² Alla tabella GDT possono essere collegate delle tabelle LDT, ovvero *local description table*, con il compito di individuare delle porzioni di memoria per conto di processi elaborativi singoli.

³ Per accesso lineare alla memoria si intende che l'indirizzo relativo del segmento corrisponde anche all'indirizzo reale della memoria stessa. In inglese si usa il termine *flat memory*.

⁴ Per rappresentare i numeri da 0 a 8191 servono precisamente 13 bit. Nei selettori di segmento si usano i 13 bit più significativi per individuare un descrittore.

Studio per un sistema a 32 bit



84.1	Introduzione a os32	241
84.1.1	Organizzazione	241
84.1.2	Le directory	243
84.1.3	La struttura degli eseguibili	243
84.1.4	Tabelle	247
84.1.5	Guida di stile	248
84.1.6	Tipi derivati speciali	252
84.2	Caricamento ed esecuzione del kernel	256
84.2.1	Multiboot	256
84.2.2	File «kernel.ld», «kernel/main/crt0.s» e «kernel/main/stack.s»	262
84.2.3	File «kernel/main.h» e «kernel/main/*»	267
84.3	Funzioni interne legate all'hardware, nei file «kernel/ibm_i386.h» e «kernel/ibm_i386/...»	274
84.3.1	Funzioni per l'input e l'output con le porte interne	274
84.3.2	Funzioni accessorie alla gestione delle interruzioni hardware	275
84.3.3	Gestione della tabella GDT	276
84.3.4	Gestione della tabella IDT	280
84.3.5	Gestione delle interruzioni	283
84.4	Gestione dei processi	286
84.4.1	File «kernel/ibm_i386/isr.s»	287

84.4.2	La tabella dei processi	298
84.4.3	Chiamate di sistema	305
84.4.4	Funzione «proc_init()»	306
84.4.5	Funzione «sysroutine()»	307
84.4.6	Funzione «proc_scheduler()»	309
84.4.7	Programmazione dei segnali	312
84.4.8	Salvataggio e recupero della pila per i «salti non locali» 318	
84.5	Caricamento ed esecuzione delle applicazioni	324
84.5.1	Caricamento in memoria	324
84.5.2	Il codice iniziale dell'applicativo	327
84.6	Gestione della memoria	332
84.6.1	File «kernel/memory.h» e «kernel/memory/...»	333
84.6.2	Scansione della mappa di memoria	337
84.7	Dispositivi	339
84.7.1	File «kernel/dev.h» e «kernel/dev/...»	340
84.7.2	File «kernel/blk.h» e «kernel/blk/...»	342
84.7.3	Numero primario e numero secondario	344
84.7.4	Dispositivi previsti	345
84.7.5	Gestione del terminale	351
84.7.6	Gestione delle unità di memorizzazione in generale 364	
84.7.7	Gestione delle unità PATA	365
84.8	Gestione del file system	370

84.8.1	File «kernel/fs/sb_...»	370
84.8.2	File «kernel/fs/zone_...»	375
84.8.3	File «kernel/fs/inode_...»	377
84.8.4	Fasi dell'innesto di un file system	389
84.8.5	Condotti	392
84.8.6	File «kernel/fs/file_...»	394
84.8.7	Descrittori di file	397
84.8.8	File «kernel/fs/path_...»	399
84.8.9	File «kernel/fs/fd_...»	405
84.9	Gestione delle interfacce di rete	409
84.9.1	Gestione dei dispositivi NE2K	409
84.9.2	Tabella delle interfacce e funzioni accessorie	412
84.9.3	Tabella ARP	417
84.10	Gestione di IPv4	420
84.10.1	Tabella IPv4	422
84.10.2	Ricezione di un pacchetto IPv4	423
84.10.3	Instradamenti	425
84.11	Gestione del protocollo ICMP	427
84.12	Gestione dei protocolli UDP e TCP	429
84.12.1	UDP	436
84.12.2	TCP	438
addr_t	252 333	
arp.h	417	
arp_clean()	418	
arp_index()	418	
arp_init()	418	
arp_reference()		

418 arp_request() 418 arp_rx()
 418 ata_cmd_identify_device()
 366 ata_cmd_read_sectors() 366
 ata_cmd_write_sectors() 366 ata_device() 366
 ata_drq() 368 ata_init() 366 ata_lba28() 368
 ata_rdy() 368 ata_read_sector() 369 ata_reset()
 366 ata_sector_t 252 365 ata_t 252 365 ata_valid()
 366 ata_write_sector() 369 blk.h 342 blk_ata.c 342
 blk_cache_init.c 342 blk_cache_read.c 342
 blk_cache_save.c 342 blk_cache_t 252 cli() 275
 crt0.s 262 dev.h 340 dev/dev_tty.c 340 DEV_CONSOLE
 345 DEV_CONSOLE **n** 345 dev_dm.c 340 DEV_DM **m n** 345
 dev_io() 340 dev_io.c 340 dev_kmem.c 340
 DEV_KMEM_ARP 345 DEV_KMEM_FILE 345
 DEV_KMEM_INODE 345 DEV_KMEM_MMP 345 DEV_KMEM_NET
 345 DEV_KMEM_PS 345 DEV_KMEM_ROUTE 345
 DEV_KMEM_SB 345 DEV_MEM 345 DEV_NULL 345 DEV_PORT
 345 DEV_TTY 345 DEV_ZERO 345 directory_t 252 dm_t
 364 fd_dup() 406 fd_reference() 406 fd_t 252
 file_reference() 397 file_stdio_dev_make() 397
 file_t 252 394 fs.h 370 gdt() 279 gdt_load() 279
 gdt_print() 279 gdt_segment() 279 gdt_t 252
 header_t 252 h_addr_t 252 420 ibm_i386.h 274
 icmp_rx() 428 icmp_tx() 428 icmp_tx_echo() 428
 icmp_tx_unreachable() 428 idt() 281 idtr_t 252
 idt_descriptor() 281 idt_irq_remap() 281
 idt_load() 281 idt_print() 281 idt_t 252
 inode_alloc() 380 inode_check() 380
 inode_dir_empty() 380 inode_file_read() 380

inode_file_write() 380 inode_free() 380
inode_fzones_read() 380 inode_fzones_write() 380
inode_get() 380 inode_pipe_make() 380
inode_pipe_read() 380 inode_pipe_write() 380
inode_print() 380 inode_put() 380
inode_reference() 380 inode_save() 380
inode_stdio_dev_make() 380 inode_t 252 377
inode_truncate() 380 inode_zone() 380 in_16() 274
in_8() 274 ip.h 420 ip_checksum() 420 ip_header() 420
ip_mask() 420 ip_reference() 420 ip_rx() 420
ip_table[] 422 ip_tx() 420 irq_off() 275 irq_on() 275
isr.s 287 isr_exception_name() 284
isr_exception_unrecoverable() 284
isr_irq_clear() 284 isr_irq_clear_pic1() 284
isr_irq_clear_pic2() 284 isr_n() 284 kbd_isr() 357
kbd_load() 357 kbd_t 252 kernel.ld 262
kernel/memory.c 333 longjmp() 318 main() 267
mboot_cmdline_opt() 261 mboot_save() 261
mb_alloc() 335 mb_alloc_size() 335 mb_clean() 335
mb_free() 335 mb_print() 335 mb_reduce() 335
mb_reference() 335 mb_size() 335 memory.h 333
MEM_BLOCK_SIZE 333 MEM_MAX_BLOCKS 333
multiboot_t 252 ne2k_check() 410 ne2k_isr() 410
ne2k_isr_expect() 410 ne2k_reset() 410 ne2k_rx() 410
ne2k_rx_reset() 410 ne2k_tx() 410 net.h 409
net_buffer_eth() 415 net_buffer_lo() 415
net_eth_ip_tx() 415 net_eth_tx() 415 net_index() 415
net_index_eth() 415 net_init() 415 net_rx() 415
os32.h 339 out_16() 274 out_8() 274

path_device() 401 path_fix() 400 path_full() 400
path_inode() 401 path_inode_link() 401 proc.h 286
proc_init() 306 proc_scheduler() 309
proc_sig_handler() 312 proc_t 252 298
route_init() 426 route_remote_to_local() 426
route_remote_to_router() 426 route_sort() 426
sb_inode_status() 373 sb_mount() 373 sb_print()
373 sb_reference() 373 sb_save() 373 sb_t 252 370
sb_zone_status() 373 screen_cell() 359
screen_clear() 359 screen_current() 359
screen_init() 359 screen_newline() 359
screen_number() 359 screen_pointer() 359
screen_putc() 359 screen_scroll() 359
screen_select() 359 screen_t 252 screen_update()
359 setjmp() 318 stack.s 262 sti() 275 sysroutine()
305 307 s_chdir() 403 s_chmod() 403 s_chown() 403 406
s_dup() 406 s_dup2() 406 s_fchmod() 406 s_fcntl()
406 s_fstat() 406 s_link() 403 s_longjmp() 318
s_lseek() 406 s_mkdir() 403 s_mknod() 403
s_mount() 403 s_open() 403 s_pipe() 406 s_read()
406 s_setjmp() 318 s_stat() 403 s_umount() 403
s_unlink() 403 s_write() 406 tcp() 433 tcp_close()
433 tcp_connect() 433 tcp_rx_ack() 433
tcp_rx_data() 433 tcp_show() 433 tcp_tx_ack() 433
tcp_tx_raw() 433 tcp_tx_rst() 433 tcp_tx_sock()
433 tty_console() 353 tty_init() 353 tty_read()
353 tty_reference() 353 tty_t 252 tty_write() 353
udp_tx() 433 zno_t 252 zone_alloc() 376
zone_free() 376 zone_print() 376 zone_read() 376

`zone_write()` 376 `__in_16()` 274 `__in_8()` 274
`__out_16()` 274 `__out_8()` 274

84.1 Introduzione a os32

os32 è uno studio che applica qualche rudimento relativo ai sistemi operativi, basandosi sull'architettura x86-32 IBM AT, utilizzando come strumenti di sviluppo GNU C, GNU AS e GNU LD, su un sistema GNU/Linux. Il risultato non è un sistema operativo utilizzabile, ma una struttura su cui poter fare esperimenti e di cui è possibile mostrare (in termini tipografici) ed eventualmente descrivere ogni riga di codice.

os32 contiene uno schedulatore banale e molto limitato, un'organizzazione dei processi ad albero e una funzionalità di amministrazione dei segnali, una gestione del file system Minix 1 e delle partizioni in stile Dos (ma solo di quelle primarie), una shell banale e qualche programma di servizio di esempio.

La differenza più importante di os32, rispetto a os16, sta nel fatto che non si utilizzano più le funzioni del BIOS tradizionale, dal momento che si opera in modalità protetta.

84.1.1 Organizzazione

Tutti i file di os32 dovrebbero essere disponibili a partire da [allegati/os32](#). In particolare, il file `'disk.hda'` è l'immagine di un vecchio disco PATA, suddiviso in due partizioni: la prima partizione con un file system Dos-FAT, contenente SYSLINUX e il kernel di os32; la seconda contenente un file system Minix 1 con il sistema.

Gli script preparati per os32 prevedono che i file system contenuti nel file-immagine che rappresentano l'unità PATA, in fase di svi-

luppo si trovino innestati nelle directory `‘/mnt/disk.hda.1/’` e `‘/mnt/disk.hda.2/’`. Pertanto, se si ricompila os32, tali directory vanno predisposte (oppure vanno modificati gli script con l'organizzazione che si preferisce attuare). Tuttavia, considerato che non è facile lavorare con file-immagine suddivisi in partizioni, altri script aiutano nelle operazioni di manutenzione: `‘fdisk’`, `‘format’`, `‘mount’`, `‘umount’`.

Per la verifica del funzionamento del sistema, è previsto l'uso equivalente di Bochs o di Qemu. Per questo scopo sono disponibili gli script `‘bochs’` e `‘qemu’` (rispettivamente i listati [94.1.2](#) e [94.1.9](#)), con le opzioni necessarie a operare correttamente.

Per la compilazione del lavoro si usano due script alternativi: `‘makeit.mer’` o `‘makeit.sep’` (listato [94.1.8](#)). Lo script `‘makeit.mer’` conduce una compilazione in cui i file eseguibili degli applicativi sono tali da condividere lo stesso segmento di memoria, sia per il codice, sia per i dati; al contrario, lo script `‘makeit.sep’` fa sì che codice e dati siano distinti (il kernel si compila solo in formato ELF). I due script ricreano ogni volta i file-make, basandosi sui file presenti effettivamente nelle varie directory previste; inoltre, alla fine della compilazione, copiano il kernel nella prima partizione del file-immagine del disco PATA (purché risulti innestata come previsto nella directory `‘/mnt/disk.hda.1/’`) e nella seconda partizione copiano gli applicativi.

Va osservato che il lavoro si basa su un file system Minix 1 (sezione [68.7](#)) perché è molto semplice, ma soprattutto, la prima versione è quella che può essere utilizzata facilmente in un sistema operativo GNU/Linux (sul quale avviene lo sviluppo di os32). È bene sottolineare che si tratta della versione con nomi da 14 caratteri, ovvero

quella tradizionale del sistema operativo Minix, mentre nei sistemi GNU/Linux, la creazione predefinita di un file system del genere produce una versione particolare, con nomi da 30 caratteri.

84.1.2 Le directory

Gli script descritti nella sezione precedente, si trovano all'inizio della gerarchia prevista per os32. Le directory successive dividono in modo molto semplice le varie componenti per la compilazione:

Directory	Contenuto
'applic/'	File delle applicazioni da usare con os32.
'kernel/'	File per la realizzazione del kernel, inclusi i file di intestazione specifici.
'lib/'	File di intestazione generali, file della libreria C per le applicazioni e, per quanto possibile, anche per il kernel.
'skel/'	Scheletro del file system complessivo, con i file di configurazione e le pagine di manuale.

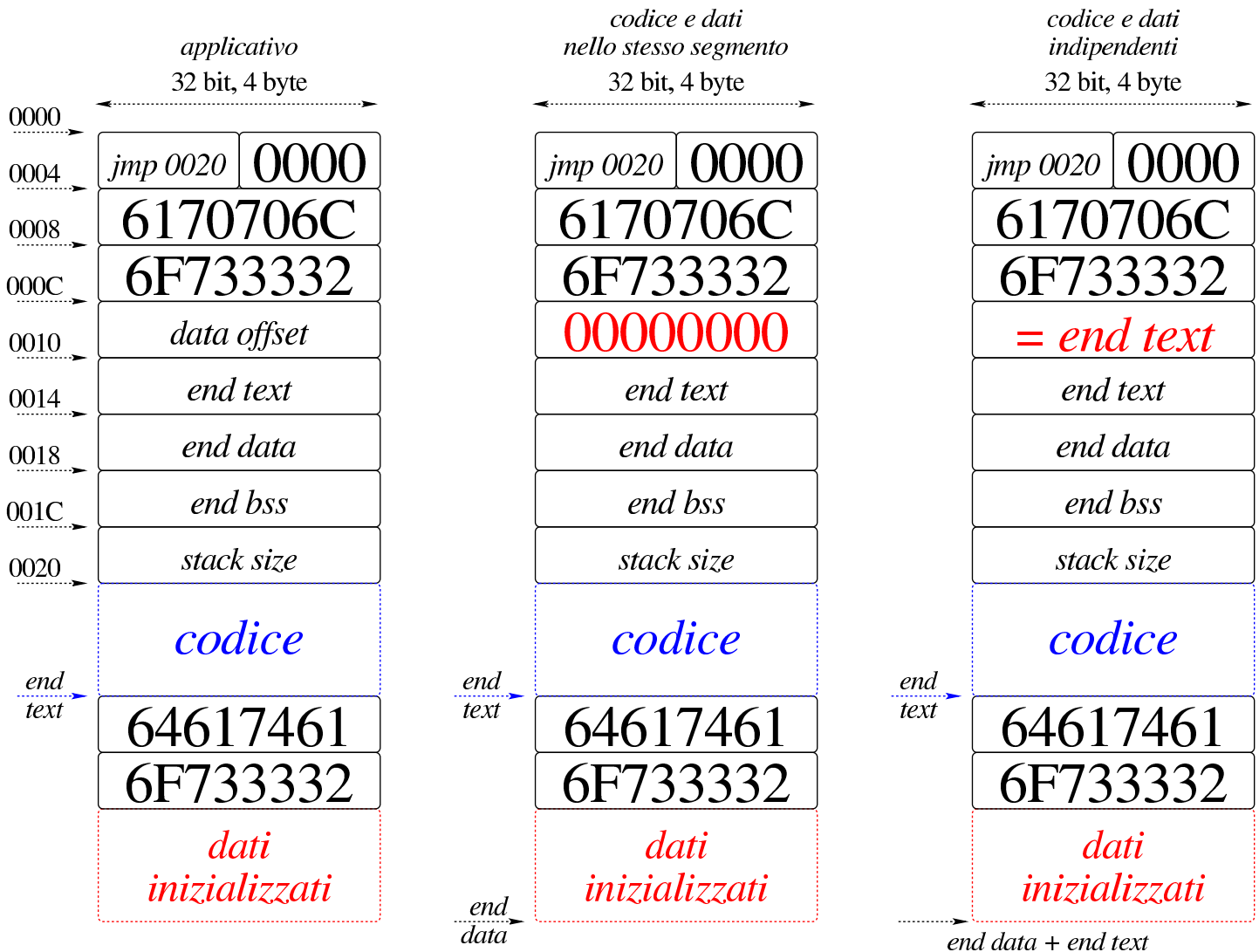
La libreria C non è completa, limitandosi a contenere ciò che serve per lo stato di avanzamento attuale del lavoro. Si osservi che nella directory 'lib/_gcc/' si collocano file contenenti una libreria di funzioni in linguaggio C, necessaria al compilatore GNU C per compiere il proprio lavoro correttamente con valori da 64 bit.

84.1.3 La struttura degli eseguibili

Nell'ottica della massima semplicità, gli eseguibili degli applicativi di os32 hanno una struttura propria, schematizzata dalla figura successiva. Tale struttura viene ottenuta attraverso i file sorgenti 'crt0.

mer.s' o 'crt0.sep.s', e i file di configurazione di GNU LD 'applic.mer.ld' o 'applic.sep.ld'. La sigla '*.mer.*' individua la compilazione in un solo segmento, sia per il codice, sia per i dati, mentre la sigla '*.sep.*' riguarda la situazione opposta, in cui codice e dati si trovano divisi.

Figura 84.2. Struttura dei file eseguibili degli applicativi di os32.



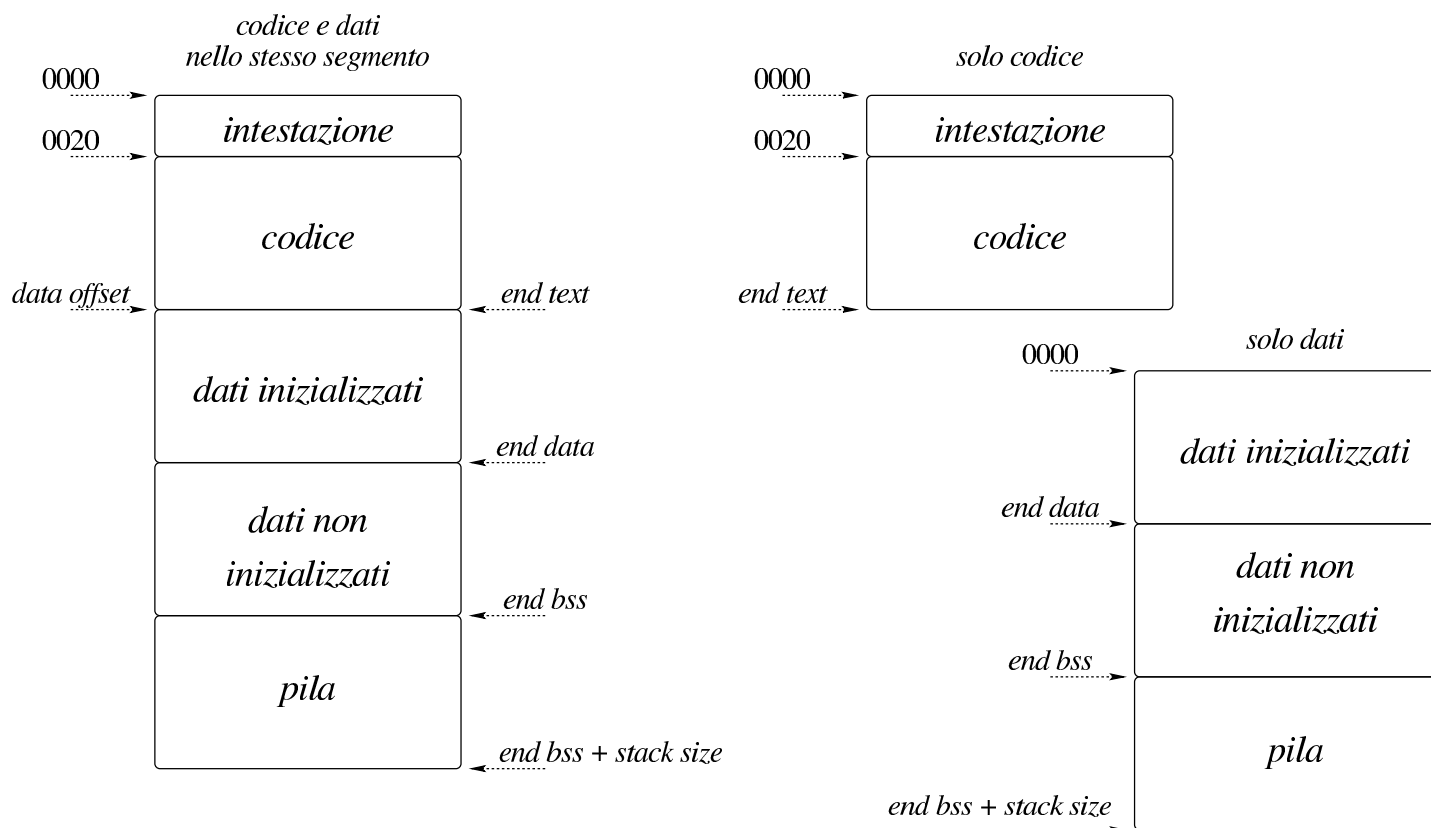
Nella figura si mettono a confronto la struttura dell'eseguibile di un applicativo compilato per avere codice e dati nello stesso segmento, rispetto al caso in cui questi sono separati. Nei primi quattro byte c'è un'istruzione di salto al codice che si trova subito dopo l'intestazione, quindi appare un'impronta di riconoscimento che occupa com-

più precisamente otto byte. Tale impronta è la rappresentazione esadecimale della stringa «os32appl», ma spezzata in due e rovesciata a causa dell'architettura *little endian* (se si legge il file in esadecimale, si vede la sequenza dei caratteri «lppa23so»).

Dopo l'impronta di riconoscimento si trovano, rispettivamente, lo scostamento del segmento dati, espresso in byte, gli indirizzi conclusivi dell'area del codice, dei dati inizializzati e di quelli non inizializzati. Alla fine viene indicata la dimensione richiesta per la pila dei dati. Per distinguere se l'eseguibile è fatto per gestire in un solo segmento codice e dati, oppure se questi devono essere separati, va osservato il valore di *data_offset*: se questo è zero, significa che il segmento dati parte dall'indirizzo zero, esattamente come il segmento codice, pertanto si trovano nello stesso spazio di indirizzamento. Se invece il valore di *data_offset* è diverso da zero, allora deve coincidere con il valore di *end_text*, in quanto i dati inizializzati si trovano nel file a partire dalla fine del codice, ma devono poi collocarsi in un segmento separato, per cui il valore di *end_data* è da intendere riferito all'indirizzo iniziale della zona dei dati, ovvero zero.

Nel file eseguibile, la porzione che contiene i dati inizializzati, parte con un'impronta ulteriore, costituita da «os32data», con gli stessi problemi di inversione già descritti per l'intestazione. Lo scopo di questa impronta è semplicemente quello di evitare che ci possano essere dati che iniziano precisamente dall'indirizzo zero, essendo questo riservato per il puntatore nullo.

Figura 84.3. Immagine in memoria dei processi generati dagli eseguibili di os32: a sinistra quelli in cui codice e dati condividono lo stesso segmento di memoria; a destra quelli che usano segmenti distinti.



Il kernel è in formato ELF, ma nella prima parte del codice viene piazzata un'impronta, secondo le specifiche multiboot (sezione [65.5.1](#)).

Riquadro 84.4. Compilazione degli applicativi con codice e dati separati.

La compilazione degli applicativi che in memoria separano il segmento codice da quello dei dati avviene in modo più complicato rispetto all'altro metodo, perché codice e dati iniziano formalmente dall'indirizzo zero e non è possibile procedere a una compilazione «binaria» normale. Pertanto, in questo caso si crea inizialmente un formato ELF provvisorio; poi, attraverso lo script 'elf-to-os32' che a sua volta si avvale di 'objdump', vengono individuate le componenti che servono dal file ELF e ricomposte secondo il formato che si aspetta os32.

84.1.4 Tabelle

Nel codice del kernel si utilizzano spesso delle informazioni organizzate in memoria in forma di tabella. Si tratta precisamente di array, le cui celle sono costituite generalmente da variabili strutturate. Queste tabelle, ovvero gli array che le rappresentano, sono dichiarate come variabili pubbliche; tuttavia, per facilitare l'accesso ai rispettivi elementi e per uniformità di comportamento, viene abbinata loro una funzione, con un nome terminante per '..._reference()', con cui si ottiene il puntatore a un certo elemento della tabella, fornendo gli argomenti appropriati. Per esempio, la tabella degli inode in corso di utilizzazione viene dichiarata così nel file 'kernel/fs/inode_table.c':

```
inode_t inode_table[INODE_MAX_SLOTS];
```

Successivamente, la funzione *inode_reference()* offre il puntatore a un certo inode:

```
inode_t *inode_reference (dev_t device, ino_t ino);
```

84.1.5 Guida di stile

«

Per cercare di dare un po' di uniformità al codice del kernel e a quello della libreria, dove possibile, i nomi delle variabili seguono una certa logica, riassunta dalla tabella successiva.

Tipo	Nome	Utilizzo
<code>inode_t *</code>	<code>inode</code> <code>inode_...</code>	Puntatore a un inode (puntatore a un elemento della tabella di inode).
<code>ino_t</code>	<code>ino</code> <code>ino_...</code>	Numero di inode, nell'ambito di un certo super blocco (ammesso che sia abbinato effettivamente a un dispositivo).
<code>int</code>	<code>fdn</code> <code>fdn_...</code>	Numero del descrittore di un file (indice all'interno della tabella dei descrittori).
<code>fd_t *</code>	<code>fd</code> <code>fd_...</code>	Puntatore a un descrittore di file (puntatore a un elemento della tabella di descrittori).
<code>int</code>	<code>fno</code> <code>fno_...</code>	Numero del file di sistema (indice all'interno della tabella dei file di sistema).
<code>zno_t</code>	<code>zone</code> <code>zone_...</code>	Numero assoluto di una «zona» del file system Minix.

Tipo	Nome	Utilizzo
zno_t	fzone fzone_...	Numero relativo di una «zona» del file system Minix. In questo caso, il numero della zona è relativo al file, dove la prima zona del file ha il numero zero.
off_t	offset offset_... off_...	Scostamento, secondo il significato del tipo derivato ' off_t '.
size_t ssize_t	size size_...	Dimensione, secondo il significato dei tipi derivati ' size_t ' o ' ssize_t '.
size_t ssize_t	count count_...	Quantità, quando il tipo ' size_t ' è appropriato.
blkcnt_t	blkcnt blkcnt_...	Quantità espressa in blocchi del file system (in questo caso, trattandosi di un file system Minix 1, si intendono zone).
blksize_t	blksize blksize_...	Dimensione del blocco del file system, espressa in byte (in questo caso, trattandosi di un file system Minix 1, si intende la dimensione della zona).

Tipo	Nome	Utilizzo
int	fno fno_...	Numero di file system.
int	oflags oflags_...	Opzioni relative all'apertura di un file, annodate nella tabella dei file di sistema: indicatori di sistema.
int	status status_...	Valore intero restituito da una funzione, quando la risposta contiene solo l'indicazione di un successo o di un insuccesso.
void *	pstatus	Puntatore restituito da una funzione, quando interessa sapere solo se si tratta di un esito valido.
char *	path path_...	Percorso del file system.
dev_t	device device_...	Numero di dispositivo, contenente sia il numero primario, sia quello secondario (<i>major</i> , <i>minor</i>).
int	n n_...	Dimensione di qualcosa, di tipo ' int '.

Tipo	Nome	Utilizzo
char *	string string_...	Area di memoria da considerare come stringa.
void *	buffer buffer_...	Area di memoria destinata ad accogliere un'informazione di tipo imprecisato.
int	n n_...	Dimensione o quantità di qualcosa, espressa attraverso il tipo ' int '.
int	c c_...	Un carattere senza segno trasformato nel tipo ' int '.
struct stat	st st_...	Variabile strutturata usata per rappresentare lo stato di un file, secondo il tipo ' struct stat '.
FILE *	fp fp_...	Puntatore che rappresenta un flusso di file.
DIR *	dp dp_...	Puntatore che rappresenta un flusso relativo a una directory.
struct dirent	dir dir_...	Variabile strutturata contenente le informazioni su una voce di una directory.

Tipo	Nome	Utilizzo
<code>struct password</code>	<code>pws</code> <code>pws_...</code>	Variabile strutturata contenente le informazioni di una voce del file <code>‘/etc/passwd’</code> .
<code>struct tm</code>	<code>tms</code> <code>tms_...</code>	Variabile strutturata contenente le componenti di un orario.
<code>struct tm *</code>	<code>timeptr</code> <code>timeptr_...</code>	Puntatore a una variabile strutturata contenente le componenti di un orario.

84.1.6 Tipi derivati speciali

«

Nel codice del kernel e nella libreria specifica di os32 si usano dei tipi derivati speciali, riassunti nella tabella successiva.

File di intestazione	Tipo speciale	Descrizione
<code>‘kernel/fs.h’</code>	<code>zno_t</code>	Variabile scalare, per rappresentare un numero di una zona, secondo la terminologia del file system Minix.
<code>‘kernel/fs.h’</code>	<code>sb_t</code>	Variabile strutturata, adatta a contenere tutte le informazioni di un super blocco, relativo a un dispositivo di memorizzazione innestato.

File di intestazione	Tipo speciale	Descrizione
'kernel/fs.h'	inode_t	Variabile strutturata, adatta a contenere tutte le informazioni di un inode aperto nel sistema.
'kernel/fs.h'	file_t	Variabile strutturata, adatta a contenere i dati di un file di sistema.
'kernel/fs.h'	fd_t	Variabile strutturata, adatta a contenere i dati di un descrittore di file, ovvero del file di un certo processo elaborativo.
'kernel/fs.h'	directory_t	Variabile strutturata, adatta a contenere una voce di una directory.
'kernel/ ibm_i386.h'	gdt_t	Variabile strutturata, adatta a contenere le informazioni di una voce della tabella GDT.
'kernel/ ibm_i386.h'	idt_t	Variabile strutturata, adatta a contenere le informazioni di una voce della tabella IDT.
'kernel/ ibm_i386.h'	idtr_t	Variabile strutturata, adatta a rappresentare il registro IDTR.

File di intestazione	Tipo speciale	Descrizione
'kernel/memory.h'	addr_t	Variabile scalare, in grado di rappresentare un indirizzo efficace di memoria (un indirizzo che vada da 00000000_{16} a $FFFFFFFF_{16}$).
'kernel/multiboot.h'	multiboot_t	Variabile strutturata che riproduce la scomposizione delle informazioni ricevute dal kernel dal sistema di avvio multiboot.
'kernel/proc.h'	proc_t	Variabile strutturata per rappresentare un elemento della tabella dei processi.
'kernel/proc.h'	header_t	Variabile strutturata che riproduce la suddivisione delle informazioni contenute nella parte iniziale degli eseguibili di os32.
'kernel/driver/ata.h'	ata_t	Variabile strutturata per rappresentare un elemento della tabella delle unità ATA.
'kernel/driver/ata.h'	ata_sector_t	Variabile strutturata per rappresentare un settore di dati relativo alla gestione ATA.

File di intestazione	Tipo speciale	Descrizione
'kernel/driver/kbd.h'	kbd_t	Variabile strutturata per rappresentare complessivamente lo stato della tastiera, incorporando anche la modalità di interpretazione corretta della mappa attuale.
'kernel/driver/screen.h'	screen_t	Variabile strutturata per rappresentare il contenuto di uno schermo e la collocazione del cursore; tale variabile strutturata compone un elemento della tabella degli schermi gestiti simultaneamente.
'kernel/driver/tty.h'	tty_t	Variabile strutturata, adatta a contenere le informazioni e lo stato di un terminale, che costituisce in pratica un elemento della tabella dei terminali.
'kernel/blk.h'	blk_cache_t	Variabile strutturata, adatta a contenere un blocco di memoria (per i dispositivi a blocchi) e le informazioni che lo riguardano, come elemento della tabella che costituisce la memoria tampone per l'accesso alle unità di memorizzazione.

File di intestazione	Tipo speciale	Descrizione
'lib/sys/os32.h'	h_addr_t	Variabile scalare a 32 bit, contenente un indirizzo IPv4 in <i>host byte order</i> , ovvero rappresentato secondo l'architettura della CPU, per ciò che riguarda l'ordine dei byte.

84.2 Caricamento ed esecuzione del kernel

«

Il kernel di os32 è compilato in formato ELF, secondo le specifiche multiboot, in modo da poter essere avviato da un sistema come GRUB 1 o SYSLINUX. Il codice del kernel inizia nel file `'crt0.s'`, dove a un certo punto viene eseguita la funzione *kmain()*, nella quale si sintetizza il funzionamento del kernel stesso. Il sistema di avvio colloca il kernel a partire dall'indirizzo 100000_{16} (1 MiByte), già in modalità protetta, di conseguenza il codice è organizzato per iniziare da tale posizione.

84.2.1 Multiboot

«

os32 è conforme alle specifiche multiboot per consentirne l'avvio attraverso GRUB 1 o SYSLINUX, senza doversi prendere carico dei problemi relativi al passaggio alla modalità protetta. Perché il file del kernel sia riconosciuto come aderente a tali specifiche, contiene un'impronta di riconoscimento, definita *multiboot header*, collocata nella parte iniziale, come dichiarato nel file `'kernel/main/crt.s'`, entro i primi 8 Kibyte.

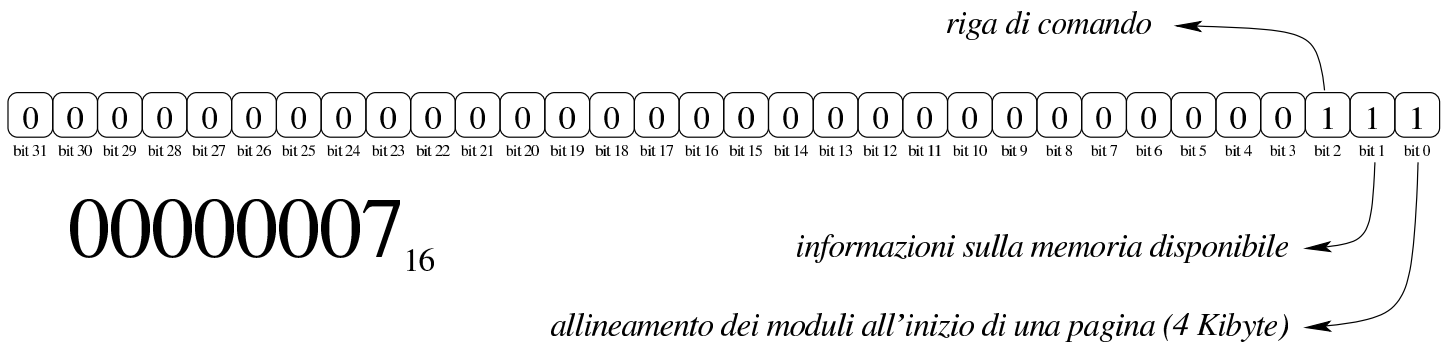
Figura 84.9. La prima parte obbligatoria dell'intestazione multiboot.



Il primo campo da 32 bit, definito *magic*, contiene l'impronta di riconoscimento vera e propria, costituita precisamente dal numero $1BADB002_{16}$. Il secondo campo da 32 bit, definito *flags*, contiene degli indicatori con i quali si richiede un certo comportamento al sistema di avvio. Il terzo campo da 32 bit, definito *checksum*, contiene un numero calcolato in modo tale che la somma tra i numeri contenuti nei tre campi da 32 bit porti a ottenere zero, senza considerare i riporti.

I nomi indicati sono quelli definiti dallo standard e, come si vede, il campo *checksum* si ottiene calcolando $-(magic + flags)$, dove si deve intendere che i calcoli avvengono con valori interi senza segno e si ignorano i riporti.

Dal momento che il kernel da avviare è in formato ELF, le informazioni che il sistema di avvio necessita per piazzarlo correttamente in memoria e per passare il controllo allo stesso, sono già disponibili e non c'è la necessità di occuparsi di altri campi facoltativi che possono seguire i tre già descritti. Stante questa semplificazione, per quanto riguarda il campo *flags*, os32 utilizza precisamente il valore 00000003_{16} , con il significato che si vede nella figura successiva.

Figura 84.10. Il campo *flags* e il suo utilizzo fondamentale.

Il bit meno significativo del campo *flags*, se impostato a uno, serve a richiedere il caricamento in memoria dei moduli eventuali (assieme al file-immagine principale) in modo che risultino allineati all'inizio di una «pagina» (ovvero all'inizio di un blocco da 4 Kibyte). os32 non prevede moduli, tuttavia richiede ugualmente questa opzione. Il secondo bit del campo *flags* serve a richiedere al sistema di avvio di passare le informazioni disponibili sulla memoria, le quali poi vengono rese disponibili a partire da un'area a cui punta inizialmente il registro *EBX*.

Figura 84.11. Calcolo del campo *checksum*.

		complemento a due	
<i>magic</i>	1BADB002+	FFFFFFFF-	1BADB009+
<i>flags</i>	<u>00000007=</u>	<u>1BADB009=</u>	<u>E4524FF7=</u>
	1BADB009	E4524FF6+	1 00000000
		<u>00000001=</u>	
<i>checksum</i>	E4524FF7		verifica della somma di controllo

Quando il sistema di avvio passa il controllo al kernel, dopo averlo caricato in memoria: il microprocessore è in modalità protetta; il registro *EAX* contiene il numero 2BADB002₁₆; il registro *EBX* contiene l'indirizzo fisico, a 32 bit, di una sequenza di campi contenenti

ti informazioni passate dal sistema di avvio (*multiboot information structure*), come si vede nella figura successiva.

Figura 84.12. Inizio della struttura di informazioni offerta da un sistema di avvio aderente alle specifiche *multiboot*.

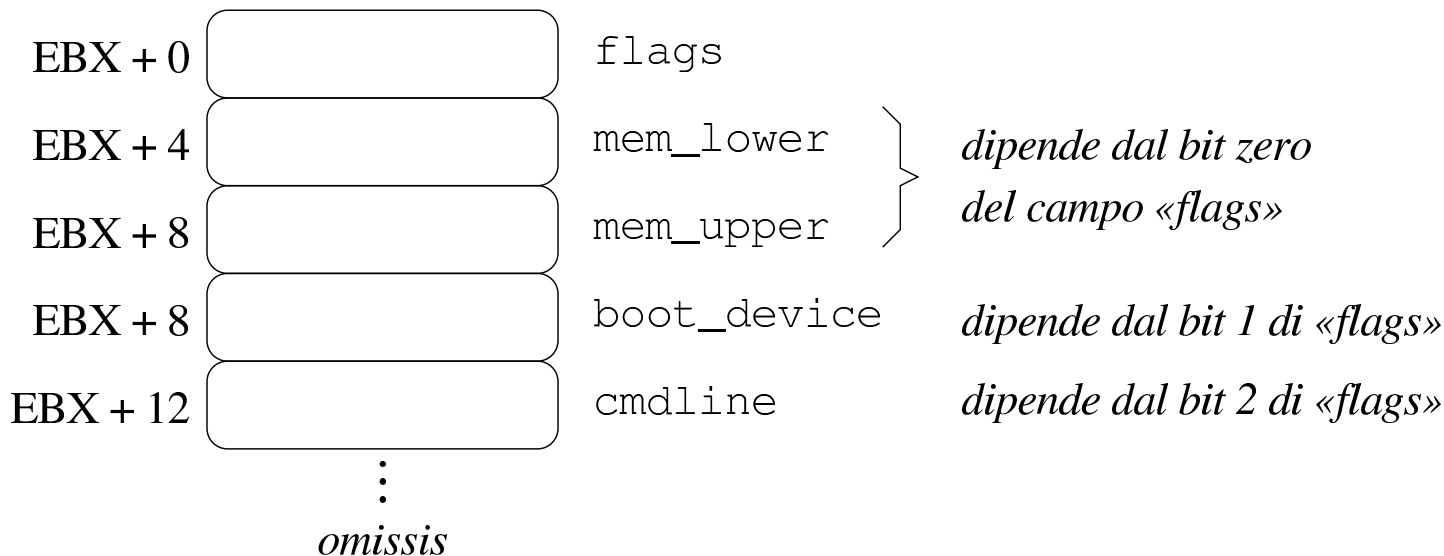


Tabella 84.13. Descrizione dei primi campi della struttura informativa fornita dal sistema di avvio *multiboot*.

Nome mnemonico del campo	bit del campo 'flags' da cui dipende	Descrizione
flags		Il primo campo definisce degli indicatori, con i quali si dichiara se una certa informazione, successiva, viene fornita ed è valida.

Nome mnemonico del campo	bit del campo 'flags' da cui dipende	Descrizione
mem_lower mem_upper	0	Se è attivo il bit meno significativo del campo 'flags', i campi 'mem_lower' e 'mem_upper' contengono la dimensione della memoria bassa (da zero a un massimo di 640 Kibyte) e della memoria alta (quella che si trova a partire da un mebibyte). La dimensione è da intendersi in kibibyte (simbolo Kibyte) e, per quanto riguarda la memoria alta, viene indicata solo la dimensione continua fino al primo «buco».
boot_device	1	Se è attivo il secondo bit, partendo dal lato meno significativo, il campo 'boot_device' dà informazioni sull'unità di avvio. L'informazione è divisa in quattro byte, come descritto nelle specifiche <i>multiboot</i> .
cmdline	2	Se è attivo il terzo bit, partendo dal lato meno significativo, il campo 'cmdline' contiene l'indirizzo iniziale di una stringa che riproduce la riga di comando passata al kernel.

Come si può intuire leggendo la tabella che descrive i primi cinque

campi, il significato dei bit del campo '**flags**' viene attribuito, mano a mano che l'aggiornamento delle specifiche prevede l'espansione della struttura informativa. Per esempio, un campo '**flags**' con il valore 100_2 sta a significare che esistono i campi fino a '**cmdline**' e il contenuto di quelli precedenti non è valido, ma i campi successivi, non esistono affatto. La comprensione di questo concetto dovrebbe rendere un po' più semplice la lettura delle specifiche.

Tabella 84.14. Funzioni per la gestione delle specifiche multiboot all'interno di os32.

Funzione	Descrizione
<pre>void mboot_save (multiboot_t *mboot_data);</pre>	<p>Salva le informazioni multiboot all'interno della variabile strutturata pubblica <i>multiboot</i>.</p> <p>Listati 94.11, 94.11.2 e 94.11.3.</p>

Funzione	Descrizione
<pre>char ** mboot_cmdline_opt (const char *opt, const char *delim);</pre>	<p>Scandisce la stringa delle opzioni salvata all'interno di <i>mboot.cmdline</i>, alla ricerca di un'opzione il cui nome corrisponda alla stringa. Dopo il nome dell'opzione deve apparire il segno '=' e dopo devono trovarsi i valori associati all'opzione, separati da <i>delim</i>. Questi valori vengono restituiti in forma di array di stringhe, dove l'ultima stringa si riconosce perché vuota.</p> <p>Listati 94.11, 94.11.2 e 94.11.1.</p>

84.2.2 File «kernel.ld», «kernel/main/crt0.s» e «kernel/main/stack.s»

«

Listati [94.1.7](#), [94.9.2](#) e [94.9.6](#).

Il codice del kernel inizia dal file 'crt.s'; tuttavia, per la sua corretta interpretazione, va considerato prima il file di configurazione di GNU LD (il collegatore, ovvero il *linker*), costituito dal file 'kernel.ld', sintetizzabile così:

```
ENTRY (kstartup)
SECTIONS {
    . = 0x00100000;
    ...
}
```

Si osserva subito che il punto di inizio del codice, descritto successivamente dal file `'crt.s'`, deve corrispondere alla posizione dell'etichetta `'kstartup'` e che quel punto deve trovarsi all'indirizzo 100000_{16} , ovvero quello in cui il sistema di avvio lo colloca.

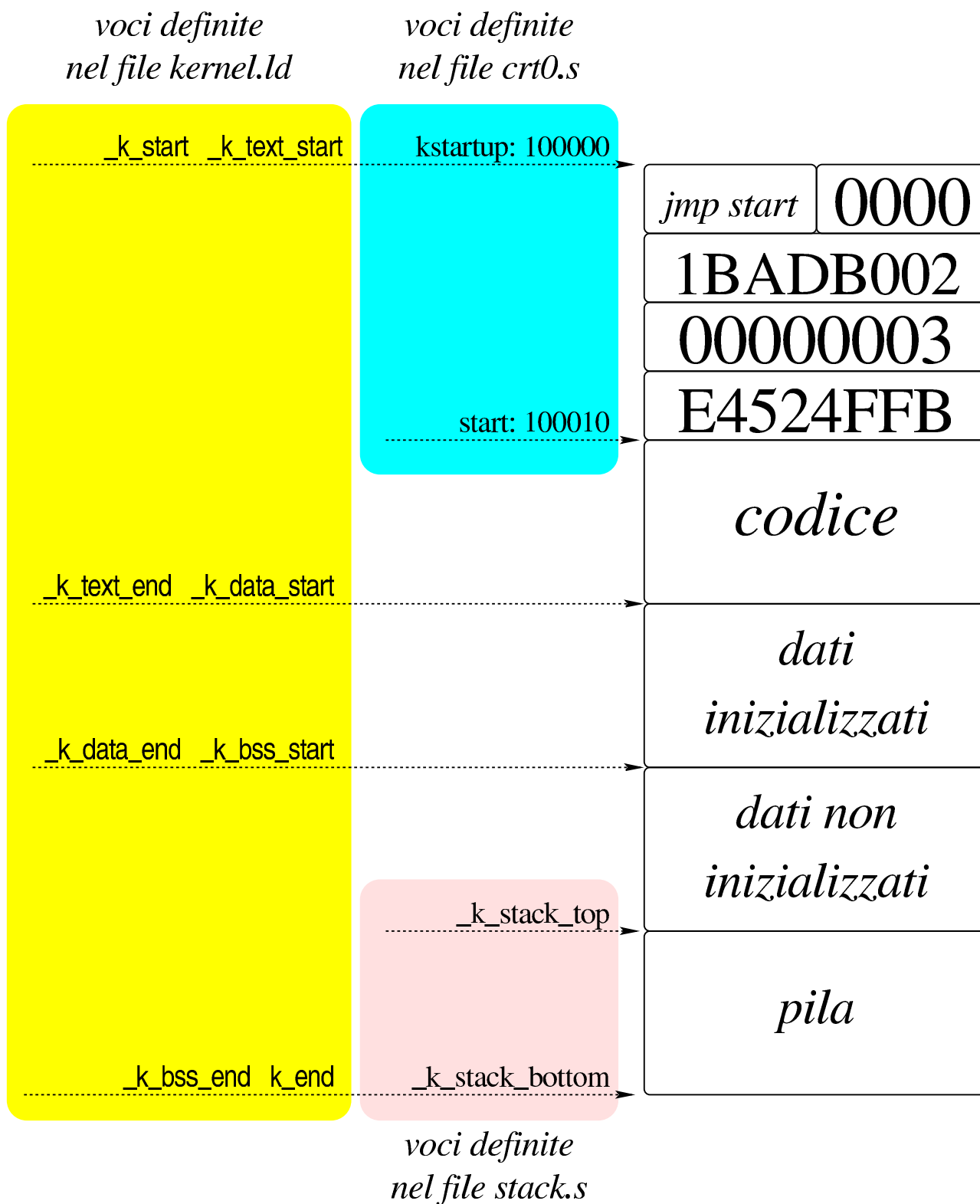
Nel file `'crt.s'`, dopo il preambolo in cui si dichiarano i simboli esterni e quelli interni da rendere pubblici, si parte proprio con l'etichetta `'kstartup'`, e da lì si salta a un'altra posizione (`'start'`), per lasciare spazio all'intestazione multiboot.

```
...
.section .text
kstartup:
    jmp start
.align 4
multiboot_header:
    .int 0x1BADB002                # magic
    .int 0x00000003                # flags
    .int -(0x1BADB002 + 0x00000003) # checksum
start:
    ...
```

L'immagine del kernel in memoria utilizza un solo segmento per codice e dati, suddividendosi nel modo consueto: codice, dati inizializzati, dati non inizializzati e pila. Per individuare le varie componenti, il file `'kernel.ld'` inserisce dei nomi a cui è possibile fare riferimento nel codice; inoltre, viene utilizzato il file `'stack.s'` per

definire lo spazio usato per la pila dei dati.

Figura 84.17. Immagine del kernel in memoria, a partire dall'indirizzo 100000_{16} , evidenziando le etichette dichiarate nei file 'kernel.ld', 'crt0.s' e 'stack.s'.



A partire dall'indirizzo corrispondente all'etichetta '**start**', nel file '`crt0.s`' inizia il lavoro preliminare del kernel. Per prima cosa viene attivata la pila dei dati, collocando nel registro *ESP* l'indirizzo corrispondente alla fine della stessa, ovvero '`_k_stack_bottom`':

```
movl $_k_stack_bottom, %esp
```

Quindi si azzerava il registro EFLAGS, sfruttando per questo la pila appena attivata:

```
pushl $0
popf
```

Infine si chiama la funzione *kmain()* (del file '`kmain.c`'), fornendo come argomenti la firma di riconoscimento del sistema multiboot, contenuta nel registro EAX, e il puntatore alla struttura contenente le informazioni fornite dal sistema multiboot, contenuto nel registro EBX:

```
pushl %ebx    # multiboot_t *info;
pushl %eax    # uint32_t      magic;
call kmain    # void kmain (uint32_t magic,
              #                multiboot_t *info);
```

Se ci dovesse essere una conclusione della funzione *kmain()*, si passerebbe al codice successivo, il quale si limita a mettere a riposo la CPU:

```
halt:
    hlt
    jmp halt
```

84.2.3 File «kernel/main.h» e «kernel/main/*»

Listato [94.9](#) e successivi.

Tutto il lavoro del kernel di os32 si sintetizza nella funzione *kmain()*, contenuta nel file ‘kernel/main/kmain.c’. Per poter dare un significato a ciò che vi appare al suo interno, occorre conoscere tutto il resto del codice, ma inizialmente è utile avere un’idea di ciò che succede, se poi si vuole compilare ed eseguire il sistema operativo.

La funzione si chiama *kmain()* (e non *main()*), perché non è conforme allo schema che dovrebbe avere la prima funzione di un programma per sistemi POSIX. Come già accennato a proposito del file ‘crt0.s’, la funzione *kmain()* prevede come parametri un codice di riconoscimento e il puntatore a delle informazioni, forniti dal sistema di avvio.

```
...
void
kmain (uint32_t magic, multiboot_t *mboot_data)
{
    ...
    tty_init ();
    if (magic == 0x2BADB002)
    {
        mboot_save (mboot_data);
        k_printf ("os32 build %s ram %i Kibyte\n",
                 BUILD_DATE, (int) multiboot.mem_upper);
        mb_size (multiboot.mem_upper * 1024);
        kbd_load ();
        blk_cache_init ();
        fs_init ();
        proc_init ();
    }
}
```

```
    }
else
    {
        ...
        k_exit ();
    }
menu ();
...
}
```

Dopo la dichiarazione delle variabili si inizializza la gestione del video della console (funzione *tty_init()*), si verifica che il codice sia stato avviato da un sistema di avvio multiboot e se ne salvano le informazioni (funzione *mboot_save()*), quindi si mostra un messaggio iniziale, si imposta la dimensione massima della memoria disponibile in base ai dati ottenuti dal sistema multiboot (funzione *mb_size()*), si configura la tastiera (funzione *kbd_load()*), si inizializza la gestione della memoria tampone (funzione *blk_cache_init()*), del file system (funzione *fs_init()*) e dei processi elaborativi (*proc_init()*). Fatto tutto questo appare un menù (funzione *menu()*) e si passa a una fase successiva.

```
...
void
kmain (uint32_t magic, multiboot_t *mboot_data)
{
    ...
    menu ();
    for (exit = 0; exit == 0;)
        {
            sys (SYS_0, NULL, 0);
            ...
            if ...
```

```
...
else if (strncmp (command, "h", MAX_CANON) == 0)
{
    menu ();
}
else if (strncmp (command, "x", MAX_CANON) == 0)
...
else if (strncmp (command, "q", MAX_CANON) == 0)
{
    k_printf ("System halted!\n");
    return;
}
}
}
```

A questo punto il kernel ha concluso le sue attività preliminari e, per motivi diagnostici, mostra un menù, quindi inizia un ciclo in cui ogni volta esegue una chiamata di sistema nulla e poi legge un comando dalla tastiera, costituito però da un solo carattere: se risulta selezionato un comando previsto, il kernel esegue quanto richiesto e poi riprende il ciclo. La chiamata di sistema nulla serve a far sì che lo schedulatore ceda il controllo a un altro processo, ammesso che questo esista, consentendo l'avvio di processi ancor prima di avere messo in funzione quel processo che deve svolgere il ruolo di **'init'**.

In generale le chiamate di sistema sono fatte per essere usate solo dalle applicazioni; tuttavia, in pochi casi speciali il kernel le deve utilizzare come se fosse proprio un'applicazione. Qui si rende necessario l'uso della chiamata nulla, perché quando è in funzione il codice del kernel non ci possono essere interruzioni esterne e quindi nessun altro processo verrebbe messo in condizione di funzionare.

Le funzioni principali disponibili in questa modalità diagnostica sono riassunte nella tabella successiva:

Comando	Risultato
h	Mostra il menù di funzioni disponibili.
t	Mostra i valori gestiti internamente dell'orologio del kernel.
f	Esegue una biforcazione del kernel, nella quale, il processo figlio si limita a mostrare ripetutamente il proprio numero di processo.
g G	Mostra le prime voci della tabella GDT, in binario.
i I	Mostra le prime voci della tabella IDT, in binario.
m	Mostra la mappa della memoria, elencando le aree continue utilizzate.
p	Mostra la situazione dei processi e altre informazioni.
s	Mostra delle informazioni sul super blocco.

Comando	Risultato
n	Mostra l'elenco degli inode attivi.
1	Invia il segnale ' SIGKILL ' al processo numero uno.
2 3 4 5 6 7 8 9 A B C D E F	Invia il segnale ' SIGTERM ' al processo con il numero corrispondente, da 2 a 15.
a b c	Avvia il programma '/bin/aaa', '/bin/bbb' o '/bin/ccc'.
x	Termina il ciclo e successivamente si passa all'avvio di '/bin/init'.
q	Ferma il sistema.

Premendo [x], il kernel avvia '/bin/init', quindi si mette in un altro ciclo, dove si limita a passare ogni volta il controllo allo schedulatore, attraverso la chiamata di sistema nulla.

```

else if (strncmp (command, "x", MAX_CANON) == 0)
{
    exec_argv[0] = "/bin/init";
    exec_argv[1] = NULL;
    pid = run ("/bin/init", exec_argv, NULL);
    while (1)
    {
        sys (SYS_0, NULL, 0);
    }
}

```

Figura 84.26. Aspetto di os32 in funzione mentre visualizza la tabella dei processi avviati e la mappa della memoria. <http://www.youtube.com/watch?v=MhS7Pv0-1qg>

```

c
abaabaaba
p
pp  p pg          T * 0x1000 D * 0x1000 stack
id id rp  tty  uid  euid  suid  usage s  addr  size  addr  size  pointer  name
 0  0  0 0000    0    0    0 00.03 R 00000 028e 00000 0000 028eb2c os32 kernel
 0  1  0 0000    0    0    0 00.09 r 0051e 000e 0052c 002d 002cf88 /bin/ccc
 1  2  0 0000   10   10   10 00.00 s 002bc 000e 002ca 002d 002cf34 /bin/aaa
 1  3  0 0000   11   11   11 00.00 s 002f7 000e 00305 002d 002cf34 /bin/bbb
ab
m
Hex mem map, blocks of 1000: 0-28f 2bc-332 51e-559
aabaab_

```


Figura 84.27. Aspetto di os32 in funzione con il menù in evidenza, dopo aver dato il comando 'x' per avviare 'init'.

```

os32 build 20AAMMGHHmm ram 130048 Kibyte
[ata_init] ATA drive 0 size 8064 Kib
[ata_drq] ERROR: drive 2 error
[dm_init] ATA drive=0 total sectors=16128
[dm_init] partition type=0c start sector=63 total sectors=2961
[dm_init] partition type=81 start sector=3024 total sectors=13104
.------.
| h      show this menu                               .------.|
| t      show internal timer values                   | all commands ||
| f      fork the kernel                             | followed by  ||
| m      memory map (HEX)                            | [Enter]      ||
| g|G    show GDT table first 21+21 items             `-----'|
| i|I    show IDT table first 21+21 items             |
| p      process status list                          |
| s      super block list                             |
| n      list of active inodes                        |
| 1..9   kill process  1 to 9                        |
| A..F   kill process 10 to 15                       |
| a..c   run programs '/bin/aaa' to '/bin/ccc' in parallel |
| x      exit interaction with kernel and start '/bin/init' |
| q      quit kernel                                  |
'-----'
x
init
os32: a basic os. [Ctrl q], [Ctrl r], [Ctrl s], [Ctrl t] to change
console.
This is terminal /dev/console0
Log in as "root" or "user" with password "ciao" :-)
login:

```

84.3 Funzioni interne legate all'hardware, nei file «kernel/ibm_i386.h» e «kernel/ibm_i386/...»

«
Listato [94.6](#) e successivi.

Il file 'kernel/ibm_i386.h' e quelli contenuti nella directory 'kernel/ibm_i386/', raccolgono il codice del kernel che è legato strettamente all'hardware, escludendo però la gestione dei dispositivi. Tra le altre spiccano particolarmente le funzioni per la gestione dei segmenti di memoria (la tabella GDT), delle interruzioni (la tabella IDT) e l'attivazione delle routine associate alle interruzioni (ISR).

Alcune delle funzioni scritte in linguaggio assembler hanno nomi che iniziano con un trattino basso, ma a fianco di queste sono disponibili delle macroistruzioni, con nomi equivalenti senza il trattino basso iniziale, per garantire che gli argomenti della chiamata abbiano il tipo corretto, restituendo un valore intero «normale», quando qualcosa deve essere restituito.

84.3.1 Funzioni per l'input e l'output con le porte interne

«
Alcune funzioni e macroistruzioni di questo gruppo sono destinate a facilitare l'input e l'output con le porte interne dell'architettura x86.

Tabella 84.28. Funzioni e macroistruzioni per l'input e l'output con le porte interne x86.

Funzione o macroistruzione	Descrizione
<code>uint32_t _in_8 (uint32_t <i>port</i>);</code>	Legge un valore a 8 bit da una porta.
<code>unsigned int in_8 (<i>port</i>);</code>	Listati 94.6 e 94.6.3 .

Funzione o macroistruzione	Descrizione
<pre>uint32_t _in_16 (uint32_t <i>port</i>); unsigned int in_16 (<i>port</i>);</pre>	<p>Legge un valore a 16 bit da una porta. Listati 94.6 e 94.6.1.</p>
<pre>void _out_8 (uint32_t <i>port</i>, uint32_t <i>value</i>); void out_8 (<i>port</i>, <i>value</i>);</pre>	<p>Scrive un valore a 8 bit in una porta. Listati 94.6 e 94.6.6.</p>
<pre>void _out_16 (uint32_t <i>port</i>, uint32_t <i>value</i>); void out_16 (<i>port</i>, <i>value</i>);</pre>	<p>Scrive un valore a 16 bit in una porta. Listati 94.6 e 94.6.4.</p>

84.3.2 Funzioni accessorie alla gestione delle interruzioni hardware

Alcune funzioni di questo gruppo sono destinate a facilitare il controllo delle interruzioni hardware che raggiungono la CPU.

Tabella 84.29. Funzioni accessorie per il controllo delle interruzioni hardware.

Funzione o macroistruzione	Descrizione
<pre>void cli (void);</pre>	<p>Disabilita le interruzioni hardware attraverso l'azzeramento dell'indicatore relativo nel registro EFLAGS. Listati 94.6 e 94.6.7.</p>

Funzione o macroistruzione	Descrizione
<pre>void sti (void);</pre>	<p>Abilita le interruzioni hardware attraverso l'attivazione dell'indicatore relativo nel registro EFLAGS.</p> <p>Listati 94.6 e 94.6.27.</p>
<pre>void irq_on (unsigned int <i>irq</i>);</pre>	<p>Abilita selettivamente l'interruzione hardware indicata per numero (da zero a 16).</p> <p>Listati 94.6 e 94.6.20.</p>
<pre>void irq_off (unsigned int <i>irq</i>);</pre>	<p>Disabilita selettivamente l'interruzione hardware indicata per numero (da zero a 16).</p> <p>Listati 94.6 e 94.6.19.</p>

84.3.3 Gestione della tabella GDT



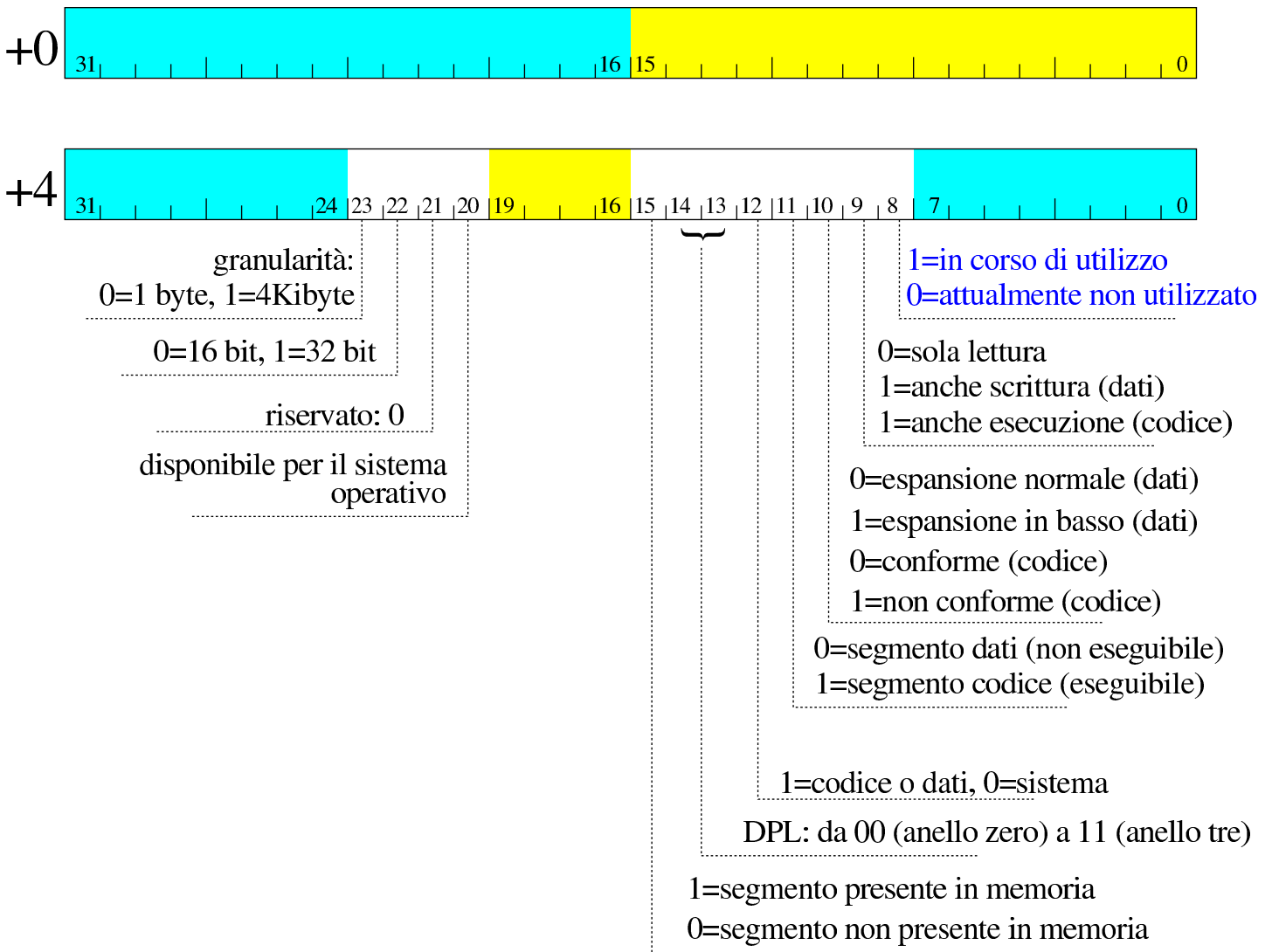
Nel momento in cui il codice del kernel prende il controllo, il microprocessore si trova già a funzionare in modalità protetta, attraverso una tabella GDT già impostata per gestire la memoria in modo lineare, senza particolari accorgimenti. Quando il kernel inizializza la gestione dei processi (funzione *proc_init()*) costruisce una nuova tabella GDT, nella quale, per ogni processo gestibile, predispone due elementi, per descrivere rispettivamente il segmento codice e il segmento dati di un processo. In pratica, la nuova tabella GDT è composta da una prima voce nulla, obbligatoria, da una coppia di voci che descrivono il segmento codice e dati del kernel, da altre cop-

pie di voci, modificate poi durante il funzionamento, per descrivere i segmenti dei processi.

Tutti i processi vedono la memoria con un indirizzamento che corrisponde a quello reale; tuttavia, disponendo ognuno di una propria coppia di voci nella tabella GDT, è possibile controllarne l'uso in modo da impedire che possano raggiungere aree al di fuori della propria competenza.

La tabella GDT è rappresentata in C dall'array ***gdt_table[]*** dichiarato nel file 'kernel/ibm_i386/gdt_public.c' (listato [94.6.11](#)), composto da elementi di tipo '***gdt_t***' (listato [94.6](#)).

```
typedef struct {
    uint32_t  limit_a          : 16,
              base_a          : 16;
    uint32_t  base_b          : 8,
              accessed        : 1,
              write_execute   : 1,
              expansion_conforming : 1,
              code_or_data    : 1,
              code_data_or_system : 1,
              dpl             : 2,
              present         : 1,
              limit_b        : 4,
              available       : 1,
              reserved        : 1,
              big             : 1,
              granularity     : 1,
              base_c          : 8;
} gdt_t;
```



La tabella viene creata con una quantità di elementi pari al valore della macro-variabile *GDT_ITEMS*. Sapendo che la prima voce è obbligatoriamente nulla, che se ne usano altre due per il kernel e che ogni processo utilizza due voci della tabella, si possono gestire al massimo $(GDT_ITEMS-3)/2$ processi.

La struttura di ogni elemento della tabella GDT è molto complessa, pertanto, per scriverci un nuovo valore si usa la funzione *gdt_segment()* che si occupa di spezzettare e ricollocare i dati come richiesto dal microprocessore (listato [94.6.12](#))

Tabella 84.32. Funzioni per la gestione della tabella GDT.

Funzione	Descrizione
<pre>void gdt_segment (int <i>segment</i>, uint32_t <i>base</i>, uint32_t <i>limit</i>, bool <i>present</i>, bool <i>code</i>, unsigned char <i>dpl</i>);</pre>	<p>Scrive una voce della tabella GDT, sezionando e ricomponendo i dati come richiesto dal microprocessore.</p> <p>Listato 94.6.12.</p>
<pre>void gdt (void);</pre>	<p>Predisporre e attivare la tabella GDT; per questo si avvale in modo particolare delle funzioni <i>gdt_segment()</i> e di <i>gdt_load()</i>.</p> <p>Listato 94.6.8.</p>
<pre>void gdt_load (void *<i>gdtr</i>);</pre>	<p>Fa sì che il microprocessore carichi la tabella GDT, a partire dal puntatore al registro GDTR; registro che contiene l'informazione della collocazione in memoria della tabella GDT e della sua estensione.</p> <p>Listato 94.6.9.</p>
<pre>void gdt_print (void *<i>gdtr</i>, unsigned int <i>first</i>, unsigned int <i>last</i>);</pre>	<p>Funzione diagnostica, usata per visualizzare il contenuto della tabella GDT in binario.</p> <p>Listato 94.6.10.</p>

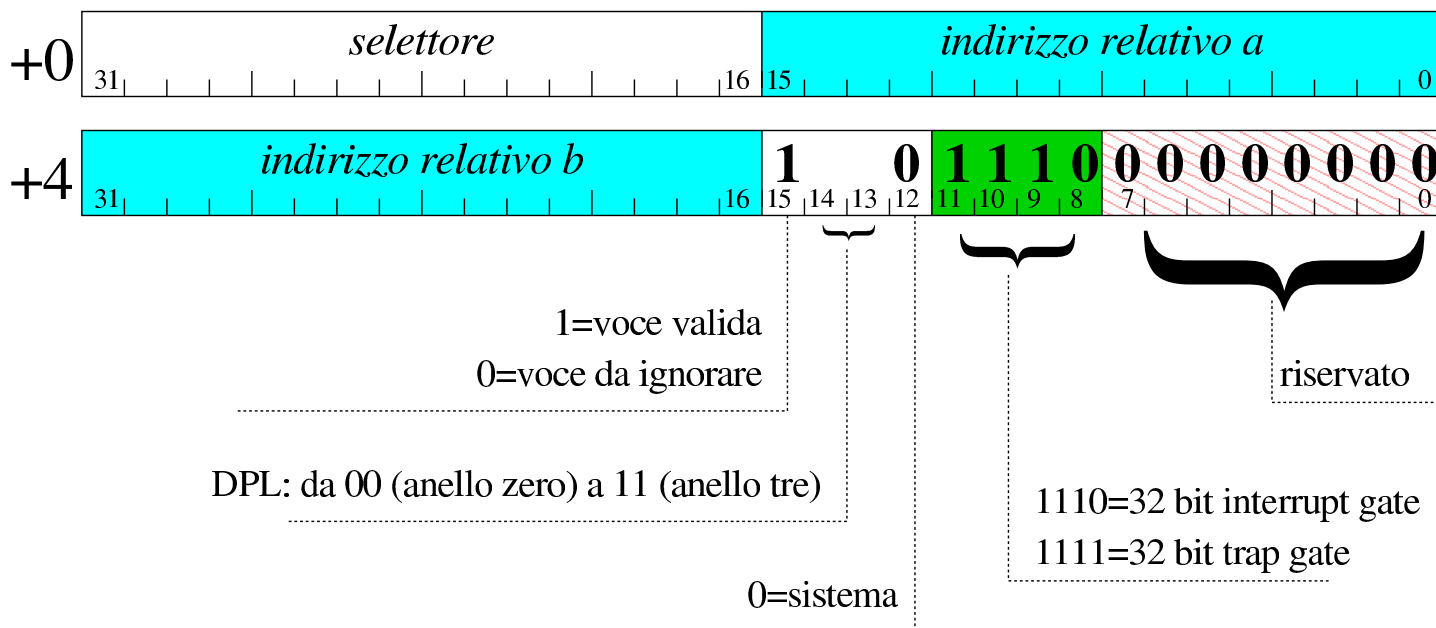
84.3.4 Gestione della tabella IDT

<<

La tabella IDT serve al microprocessore per conoscere quali procedure avviare al verificarsi delle interruzioni. La funzione *idt()* si occupa di predisporre la tabella e di attivarla, ma prima di ciò si prende cura di posizionare le interruzioni hardware a partire dalla voce 32 (la 33-esima). Le procedure a cui fa riferimento la tabella IDT creata con la funzione *idt()* sono dichiarate nel file ‘kernel/ibm_i386/isr.s’, descritto però nella sezione successiva.

La tabella IDT è rappresentata in C dall’array *idt_table[]* dichiarato nel file ‘kernel/ibm_i386/idt_public.c’ (listato [94.6.18](#)), composto da elementi di tipo ‘*idt_t*’ (listato [94.6](#)).

```
typedef struct {
    uint32_t  offset_a : 16,
              selector : 16;
    uint32_t  filler   : 8,
              type     : 4,
              system   : 1,
              dpl      : 2,
              present  : 1,
              offset_b : 16;
} idt_t;
```

La tabella viene creata con 129 elementi, anche se più della metà non vengono usati; tuttavia, proprio l'ultimo, corrispondente all'interruzione 128_{10} , ovvero 80_{16} , serve per le chiamate di sistema.

La struttura di ogni elemento della tabella IDT è un po' complicata, pertanto, per scriverci un nuovo valore si usa la funzione *idt_descriptor()* che si occupa di spezzettare e ricollocare i dati come richiesto dal microprocessore (listato [94.6.14](#))

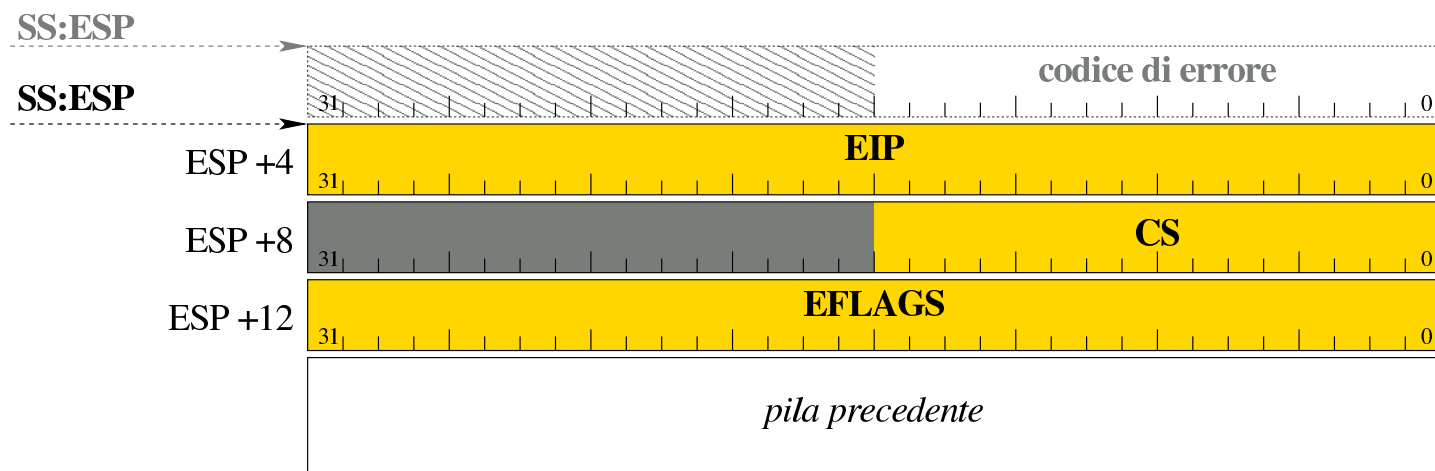
Tabella 84.35. Funzioni per la gestione della tabella IDT.

Funzione	Descrizione
<pre>void idt_descriptor (int <i>desc</i> , void *<i>isr</i> , uint16_t <i>selector</i> , bool <i>present</i> , char <i>type</i> , char <i>dpl</i>) ;</pre>	<p>Scrive una voce della tabella IDT, sezionando e ricomponendo i dati come richiesto dal microprocessore.</p> <p>Listato 94.6.14.</p>

Funzione	Descrizione
<pre>void idt (void);</pre>	<p>Predisporre e attivare la tabella IDT; per questo si avvale in modo particolare delle funzioni <i>idt_descriptor()</i> e di <i>idt_load()</i>.</p> <p>Listato 94.6.13.</p>
<pre>void idt_load (void *<i>idtr</i>);</pre>	<p>Fa sì che il microprocessore carichi la tabella IDT, a partire dal puntatore al registro IDTR; registro che contiene l'informazione della collocazione in memoria della tabella IDT e della sua estensione.</p> <p>Listato 94.6.16.</p>
<pre>void idt_irq_remap (unsigned int <i>offset_1</i>, unsigned int <i>offset_2</i>);</pre>	<p>Modifica la mappatura delle interruzioni hardware (IRQ) spostando il primo gruppo a partire dal valore di <i>offset_1</i> e il secondo gruppo a partire da <i>offset_2</i>.</p> <p>Listato 94.6.15.</p>
<pre>void idt_print (void *<i>idtr</i>, unsigned int <i>first</i> unsigned int <i>last</i>);</pre>	<p>Funzione diagnostica, usata per visualizzare il contenuto della tabella IDT in binario.</p> <p>Listato 94.6.17.</p>

84.3.5 Gestione delle interruzioni

Le interruzioni che individua il microprocessore (eccezioni, interruzioni software e interruzioni hardware) fanno interrompere l'attività normale dello stesso, costringendolo ad accumulare nella pila attuale dei dati lo stato di alcuni registri ed eventualmente di un codice di errore, saltando poi alla posizione di codice indicata nella voce corrispondente nella tabella IDT. Va osservato che, per semplicità, os32 fa lavorare i propri processi nell'anello zero, come il kernel, per cui i dati accumulati nella pila si limitano a quelli della figura successiva, perché non c'è mai un passaggio da un livello di privilegio a un altro.



Le posizioni del codice a cui il microprocessore deve saltare, secondo le indicazioni della tabella IDT, sono contenute tutte nel file 'kernel/ibm_i386/isr.s', mentre nel file 'kernel/ibm_i386.h' vi si fa riferimento attraverso dei prototipi di funzione, benché non si tratti propriamente di funzioni.

Tabella 84.37. Funzioni per la gestione delle interruzioni.

Funzione	Descrizione
<pre>void isr_<i>n</i> (void);</pre>	<p>Si tratta di procedure attivate dalle interruzioni, dove per esempio <i>isr_33()</i> viene eseguita a seguito del verificarsi dell'interruzione numero 33, la quale ha origine da IRQ 1, ovvero dalla tastiera. L'indicazione di quale procedura attivare per ogni interruzione dipende dalla configurazione della tabella IDT.</p> <p>Listato 94.6.21.</p>

Funzione	Descrizione
<pre> void isr_exception_unrecoverable (uint32_t <i>eax</i>, uint32_t <i>ecx</i>, uint32_t <i>edx</i>, uint32_t <i>ebx</i>, uint32_t <i>ebp</i>, uint32_t <i>esi</i>, uint32_t <i>edi</i>, uint32_t <i>ds</i>, uint32_t <i>es</i>, uint32_t <i>fs</i>, uint32_t <i>gs</i>, uint32_t <i>interrupt</i>, uint32_t <i>error</i>, uint32_t <i>eip</i>, uint32_t <i>cs</i>, uint32_t <i>eflags</i>); </pre>	<p>Questa funzione viene usata all'interno del file 'isr.s' per segnalare il verificarsi di un'eccezione non risolvibile, come nel caso di una divisione per zero. Pertanto, la funzione ha soprattutto un significato diagnostico. Listato 94.6.23.</p>
<pre> char *isr_exception_name (int <i>exception</i>); </pre>	<p>Restituisce il puntatore alla stringa contenente il nome dell'eccezione corrispondente al numero di interruzione fornito. Viene usata da <i>isr_exception_unrecoverable()</i> per dare delle indicazioni comprensibili sull'eccezione che si è verificata. Listato 94.6.22.</p>

Funzione	Descrizione
<pre>void isr_irq_clear (uint32_t <i>idtn</i>);</pre>	<p>Avvisa il PIC (<i>programmable interrupt controller</i>) che l'interruzione hardware emessa è stata recepita e se ne possono ricevere altre. Tuttavia, essendoci due PIC, la funzione stabilisce quale dei due è coinvolto direttamente e di conseguenza come procedere. Listato 94.6.24.</p>
<pre>void isr_irq_clear_pic1 (void);</pre>	<p>Avvisa il PIC1 che l'interruzione hardware emessa è stata recepita e se ne possono ricevere altre. Listato 94.6.25.</p>
<pre>void isr_irq_clear_pic2 (void);</pre>	<p>Avvisa il PIC2 che l'interruzione hardware emessa è stata recepita e se ne possono ricevere altre. Listato 94.6.26.</p>

84.4 Gestione dei processi



Listato [94.6.21](#); listato [94.14](#) e successivi.

La gestione dei processi è raccolta nei file 'kernel/proc.h' e 'kernel/proc/...'; tuttavia, dal file 'kernel/ibm_i386/isr.s' hanno origine le procedure attivate dalle interruzioni e dalle chiama-

te di sistema: le chiamate di sistema e le interruzioni provenienti dal temporizzatore interno provocano l'attivazione dello schedatore.

Con `os32`, quando un processo viene interrotto per lo svolgimento del compito dell'interruzione, si passa sempre a utilizzare la pila dei dati del kernel. Per annotare la posizione in cui si trova l'indice della pila del kernel si usa la variabile `_ksp`, accessibile anche dal codice in linguaggio C.

Il codice del kernel può essere interrotto dagli impulsi del temporizzatore, ma in tal caso non viene coinvolto lo schedatore per lo scambio con un altro processo, così che dopo l'interruzione è sempre il kernel che continua a funzionare; pertanto, nella funzione `kmain()` è il kernel che cede volontariamente il controllo a un altro processo (ammesso che ci sia) con una chiamata di sistema nulla.

84.4.1 File «kernel/ibm_i386/isr.s»

Il file `'kernel/ibm_i386/isr.s'` contiene il codice per la gestione delle interruzioni dei processi. Nella parte iniziale del file, vengono dichiarate delle variabili, alcune delle quali sono pubbliche e accessibili anche dal codice in C.

```
.section .data
proc_syscallnr:      .int    0x00000000
proc_msg_offset:    .int    0x00000000
proc_msg_size:      .int    0x00000000
proc_instruction_pointer: .int    0x00000000
proc_back_address:  .int    0x00000000
_ksp:               .int    0x00000000
syscall_working:    .int    0x00000000
_clock_kernel:
kticks_lo:          .int    0x00000000
kticks_hi:          .int    0x00000000
```

```
_clock_time:  
tticks_lo:          .int    0x00000000  
tticks_hi:          .int    0x00000000
```

Si tratta di variabili scalari da 32 bit, tenendo conto che: i simboli `'kticks_lo'` e `'kticks_hi'` compongono assieme la variabile `_clock_kernel` a 64 bit per il linguaggio C; i simboli `'tticks_lo'` e `'tticks_hi'` compongono assieme la variabile `_clock_time` a 64 bit per il linguaggio C.

Dopo la dichiarazione delle variabili inizia il codice vero e proprio, dove i simboli `'isr_n'` si riferiscono al codice da usare in presenza dell'interruzione *n*. Tra tutte, le interruzioni più importanti sono quelle del temporizzatore (`isr_32()`), il quale produce un impulso a circa 100 Hz; quelle della tastiera (`isr_33()`) e delle chiamate di sistema (`isr_128()`).

Il codice per la gestione dei tre tipi di interruzione più importanti ha delle similitudini che conviene analizzare simultaneamente. `os32` non cambia mai anello, nel senso che il livello di privilegio dei processi è pari a quello del kernel; pertanto, nel momento in cui si verifica un'interruzione, la pila e il segmento dati in essere sono quelli del processo interrotto. Le procedure che gestiscono le tre interruzioni principali iniziano con il salvataggio dei registri nella pila attuale e il passaggio al segmento dei dati del kernel, lasciando temporaneamente la pila nel segmento dati del processo interrotto; nello stesso modo, terminano con il ripristino del segmento dati originario (al momento dell'interruzione) e il ripristino successivo dei registri, estraendone i valori dalla pila:

```
#
```



```
# Save into process stack:  
#  
pushl %gs  
pushl %fs  
pushl %es  
pushl %ds  
pushl %edi  
pushl %esi  
pushl %ebp  
pushl %ebx  
pushl %edx  
pushl %ecx  
pushl %eax  
#  
# Set the data segments to the kernel data segment,  
# so that the following variables can be accessed.  
#  
mov  $16, %ax # DS, ES, FS and GS.  
mov  %ax, %ds  
mov  %ax, %es  
mov  %ax, %fs  
mov  %ax, %gs  
...  
...  
#  
# Restore from process stack.  
#  
popl %eax  
popl %ecx  
popl %edx  
popl %ebx  
popl %ebp  
popl %esi  
popl %edi
```

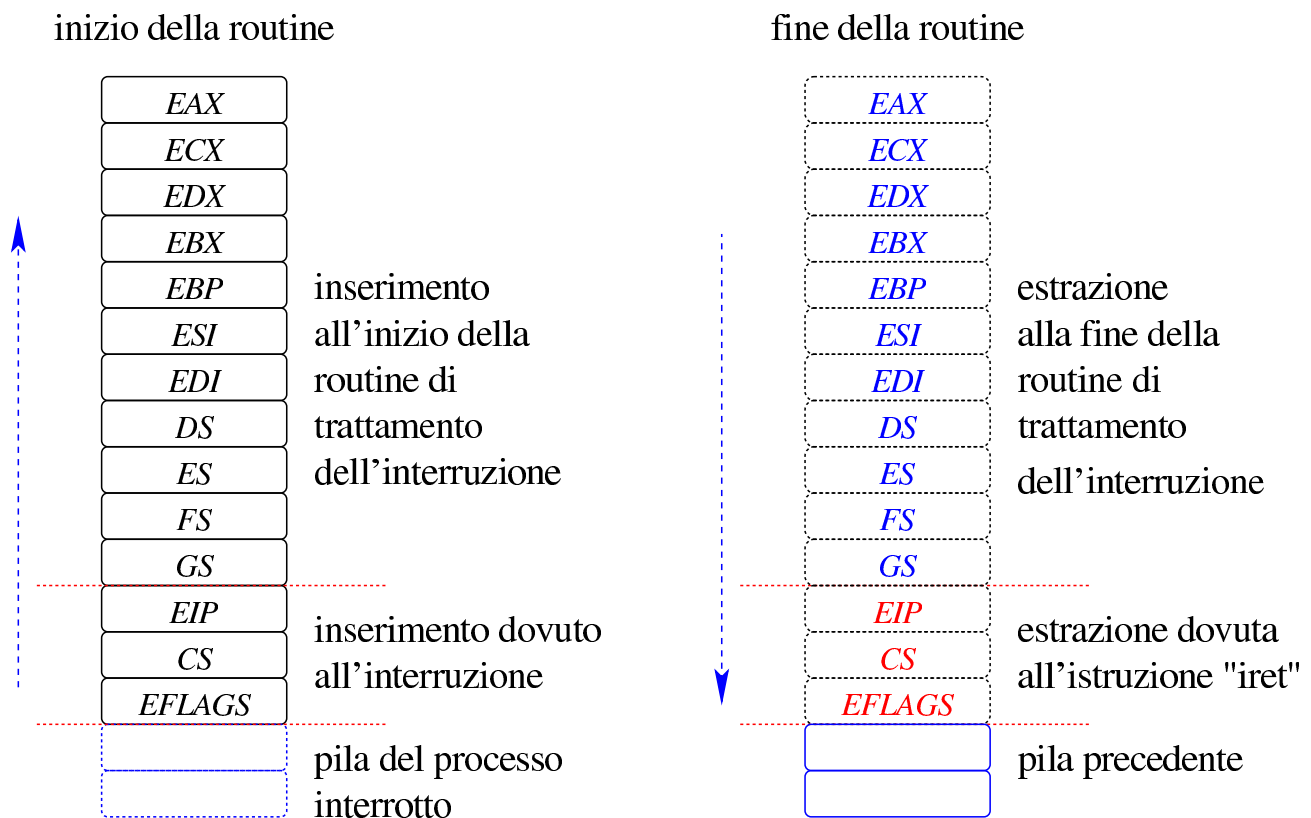
```

popl %ds
popl %es
popl %fs
popl %gs
...
iret

```

Il segmento dati del kernel si trova nella terza voce della tabella GDT (la prima è nulla, la seconda è per il codice del kernel, la terza è per i dati del kernel). Sapendo che ogni voce occupa 8 byte (64 bit), per raggiungere l'inizio della terza voce occorre indicare il valore 16 nel registro di segmento.

Figura 84.40. Inserimento nella pila del processo interrotto.



Durante l'elaborazione di un'interruzione proveniente dal temporizzatore o dalla tastiera, è necessario sapere se è già in corso l'elaborazione di una chiamata di sistema. Se ciò accade, l'impulso del tem-

porizzatore viene recepito, incrementando i contatori, ma non viene fatto altro, mentre l'impulso della tastiera viene semplicemente ignorato.

```
#
# Check if a system call is already working: if so,
# just leave (go to L2).
#
cml $1, syscall_working
je L2
```

In pratica, quando si presenta una chiamata di sistema, inizialmente viene assegnato il valore uno alla variabile *syscall_working*, mentre alla fine del suo compito questa variabile viene azzerata:

```
#
# Tell that it is a system call.
#
movl $1, syscall_working
...
#
# End of system call.
#
movl $0, syscall_working
```

Quando l'interruzione proviene dal temporizzatore e non è in corso l'esecuzione di una chiamata di sistema, oppure quando l'interruzione deriva proprio da una chiamata di sistema, viene attivato lo schedulatore (direttamente o indirettamente, attraverso la funzione che svolge il lavoro richiesto dalla chiamata di sistema), ma per fare questo, è necessario passare alla pila dei dati del kernel, per poi ripristinarla successivamente:

```
#
```

```
# Save process stack registers into kernel data segment.  
#  
mov %ss,  proc_stack_segment_selector  
mov %esp, proc_stack_pointer  
...  
#  
# Switch to kernel stack.  
#  
mov $16, %ax  
mov %ax, %ss  
mov _ksp, %esp  
...  
...  
#  
# Restore process stack registers from kernel data  
# segment.  
#  
mov proc_stack_segment_selector, %ss  
mov proc_stack_pointer, %esp
```

Figura 84.44. Scambi delle pile: prima fase.

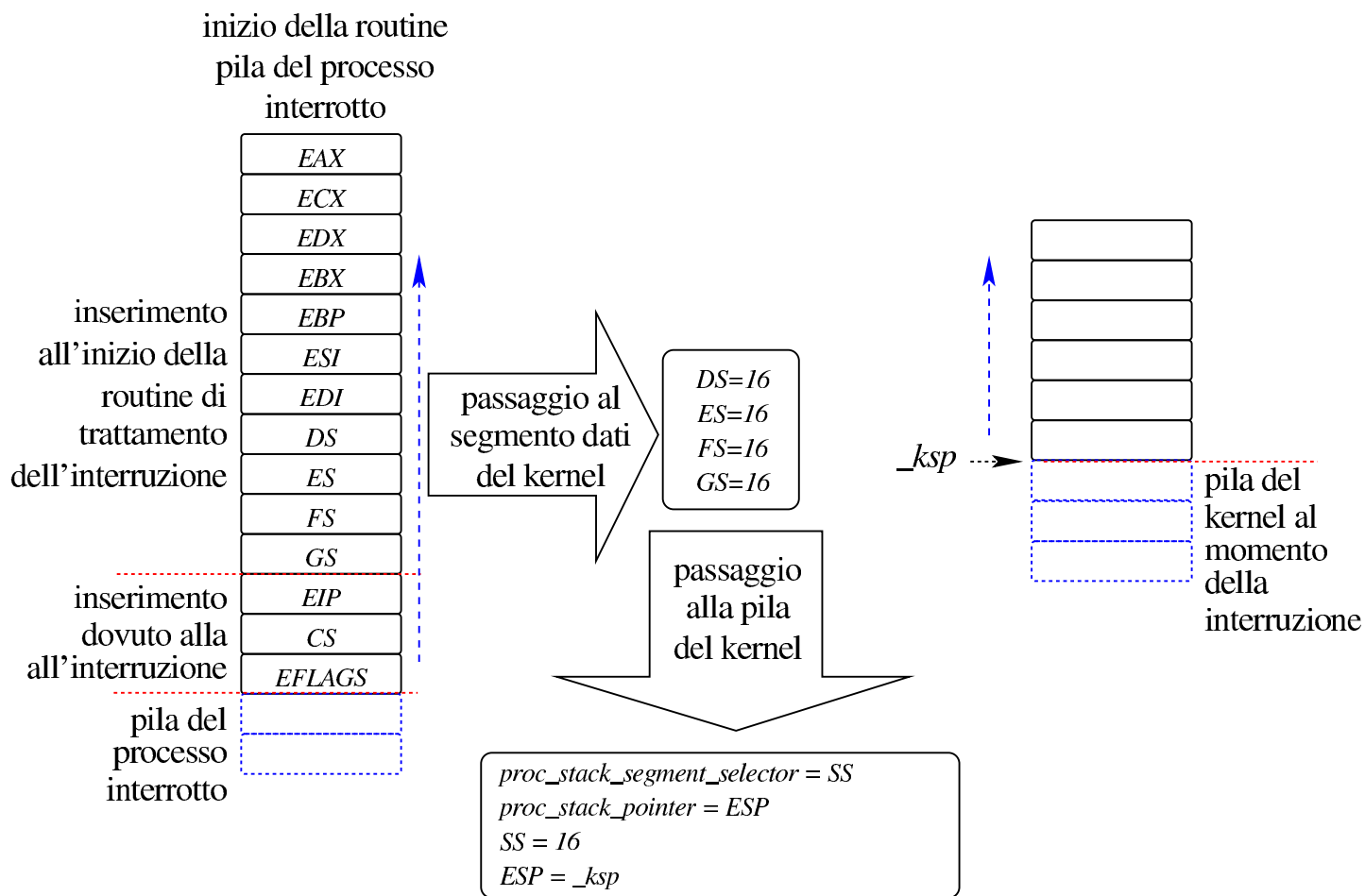
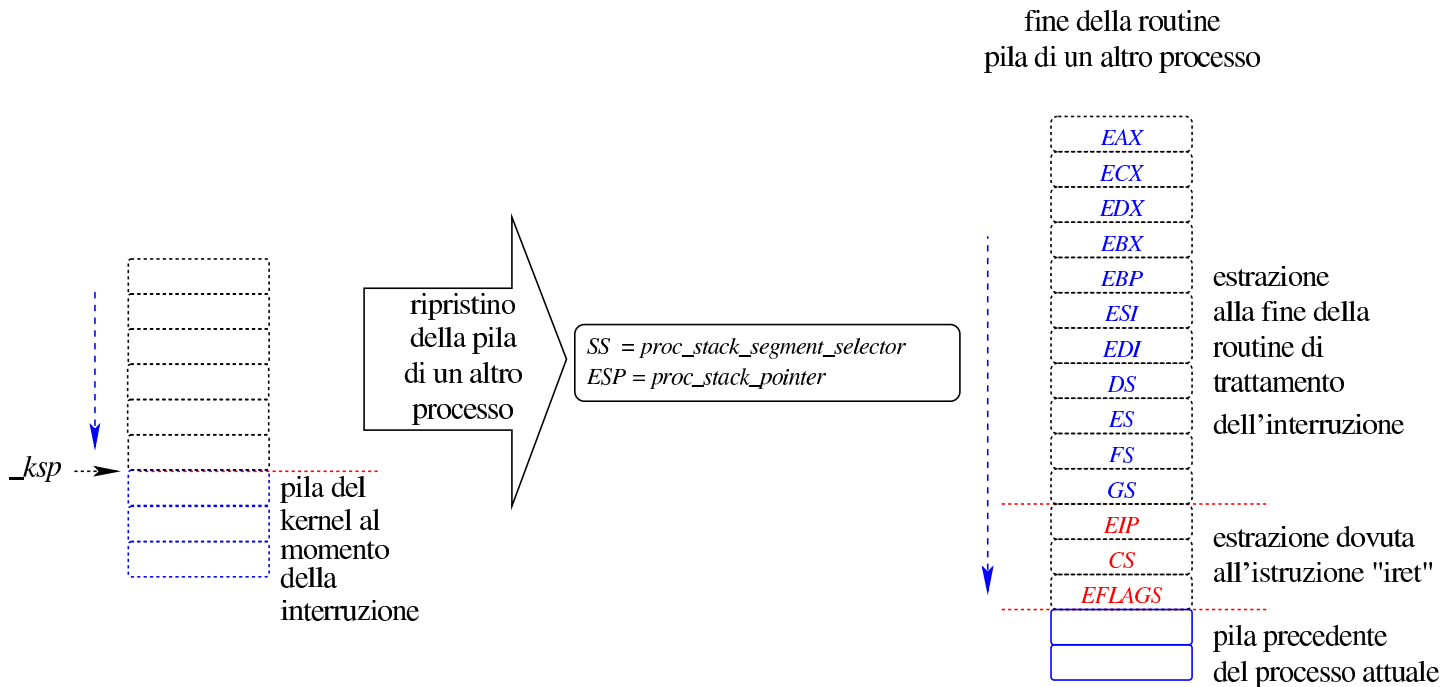


Figura 84.45. Scambi delle pile: seconda fase.



84.4.1.1 Particolarità della routine «isr_32», ovvero «irq_timer»

«

Dopo il salvataggio dei registri principali e dopo il cambiamento del segmento dati, rimanendo ancora sulla pila dei dati del processo interrotto, la routine di gestione delle interruzioni del temporizzatore si occupa di incrementare i contatori degli impulsi. Gli impulsi giungono alla frequenza di 100 Hz circa, per cui non c'è la necessità di fare alcun tipo di conversione:

```

...
isr_32:          # IRQ 0: «timer»
    cli
    jmp irq_timer
...
irq_timer:
    ...
    add $1, kticks_lo    # Kernel ticks counter.

```

```

adc $0, kticks_hi      #
#
add $1, tticks_lo     # Clock ticks counter.
adc $0, tticks_hi     #
...

```

A questo punto, se l'interruzione è avvenuta mentre era in corso l'elaborazione di una chiamata di sistema, tutto si conclude con il ripristino dei registri e del PIC1, in modo da consentire la ripresa delle interruzioni. Se invece l'interruzione è avvenuta in una situazione differente, si verifica ancora che non sia stato interrotto il funzionamento del kernel stesso, perché se così fosse, anche in questo caso la procedura termina con il solito ripristino dei registri e del PIC1.

```

#
# Check if it is already in kernel mode: the kernel has
# PID 0. If so, just leave (go to L2).
#
mov proc_current, %edx      # Interrupted PID.
mov $0,             %eax    # Kernel PID.
cmp %eax, %edx
je L2

```

Se non è stato interrotto il codice del kernel, viene chiamata la funzione *proc_scheduler()*, la quale può cambiare i valori delle variabili pubbliche *proc_stack_segment_selector* e *proc_stack_pointer*, provocando così la sostituzione del processo interrotto, quando subito dopo si ripristina la pila a cui queste due variabili fanno riferimento.

84.4.1.2 Particolarità della routine «`isr_128`»

<<

La pila dei dati al momento dell'interruzione dovuta a una chiamata di sistema, contiene anche le informazioni necessarie a conoscere il tipo di funzione richiesta e gli argomenti di questa, in forma di variabile strutturata, di cui viene trasmesso il puntatore.

Dopo il salvataggio dei registri principali e dopo il cambiamento del segmento dati, rimanendo ancora sulla pila dei dati del processo interrotto, si recuperano dalla pila le informazioni necessarie a ricostruire la funzione richiesta, salvandole in variabili locali:

```
#
# Save some more data, from the system call.
#
.equ SYSCALL_NUMBER,      60
.equ MESSAGE_OFFSET,     64
.equ MESSAGE_SIZE,       68
#
mov %esp, %ebp
mov SYSCALL_NUMBER(%ebp), %eax
mov %eax, proc_syscallnr
mov MESSAGE_OFFSET(%ebp), %eax
mov %eax, proc_msg_offset
mov MESSAGE_SIZE(%ebp), %eax
mov %eax, proc_msg_size
```

A questo punto, in modo simile a quanto avviene per le interruzioni del temporizzatore, si verifica se la chiamata di sistema è avvenuta durante il funzionamento del kernel, cosa che os32 consente. Tuttavia, la chiamata di sistema viene eseguita ugualmente, solo che si salva l'indice della pila nella variabile *`_ksp`*; pertanto, è proprio attraverso una prima chiamata di sistema nulla che os32 inizializza la

gestione delle interruzioni.

```

#
# Check if it is already in kernel mode: the kernel has
# PID 0.
#
mov proc_current, %edx      # Interrupted PID.
mov $0,          %eax      # Kernel PID.
cmp %eax, %edx
jne L3
#
mov %esp, _ksp
L3:

```

A questo punto viene eseguito il passaggio alla pila del kernel, indipendentemente dal fatto che serva o meno, quindi viene chiamata la funzione *sysroutine()*, inserendo nella pila attuale i parametri richiesti e salvati precedentemente all'interno di variabili locali:

```

push proc_msg_size
push proc_msg_offset
push proc_syscallnr
call sysroutine
add $4, %esp
add $4, %esp
add $4, %esp

```

I passi successivi includono il ripristino della pila precedente, secondo quanto annotato nelle variabili globali *proc_stack_segment_selector* e *proc_stack_pointer*, e lo stato dei registri dalla nuova pila.

Va osservato che la funzione *sysroutine()* oltre che prendersi carico di eseguire il compito della chiamata di sistema richiesta, provvede poi a sostituire il processo interrotto, avvalendosi a sua volta della

funzione *proc_scheduler()*.

84.4.2 La tabella dei processi

«

Listato 94.14.

Nel file ‘kernel/proc.h’ viene definito il tipo ‘**proc_t**’, con il quale, nel file ‘kernel/proc/proc_public.c’ si definisce la tabella dei processi, rappresentata dall’array *proc_table[]*.

Listato 84.51. Struttura del tipo ‘**proc_t**’, corrispondente agli elementi dell’array *proc_table[]*.

```
typedef struct {
    pid_t      ppid;           // Parent PID.
    pid_t      pgrp;          // Process group ID.
    uid_t      uid;           // Real user ID
    uid_t      euid;          // Effective user ID.
    uid_t      suid;          // Saved user ID.
    gid_t      gid;           // Real group ID
    gid_t      egid;          // Effective group ID.
    gid_t      sgid;          // Saved group ID.
    dev_t      device_tty;    // Controlling terminal.
    char       path_cwd[PATH_MAX];
                                // Working directory path.
    inode_t    *inode_cwd;    // Working directory inode.
    int        umask;         // File creation mask.
    unsigned long int sig_status; // Active signals.
    unsigned long int sig_ignore; // Signals to be ignored.
    uintptr_t  sig_handler[MAX_SIGNALS];
                                // Opt. sig. handlers.
    uintptr_t  sig_handler_wrapper;
                                // Special wrapper.
    clock_t    usage;         // Clock ticks CPU
                                // time usage.
    unsigned int status;
    int        wakeup_events; // Wake up for something.
}
```

```

int                wakeup_signal; // Signal waited.
unsigned int      wakeup_timer;  // Seconds to wait for.
inode_t           *wakeup_inode; // Inode waited.
addr_t            address_text;
size_t            domain_text;
addr_t            address_data;
size_t            domain_data;
size_t            domain_stack; // Included inside the
                               // data.
size_t            extra_data;    // Extra data for 'brk()'.
uint32_t          sp;
int               ret;
char              name[PATH_MAX];
fd_t              fd[FOPEN_MAX];
} proc_t;

```

La tabella successiva descrive il significato dei vari membri previsti dal tipo `proc_t`. Va osservato che la cosiddetta «u-area» (*user area*) non viene gestita come un sistema Unix tradizionale e tutti i dati dei processi sono raccolti nella tabella gestita dal kernel. Di conseguenza, dal momento che i processi non dispongono di una tabella personale con i dati della u-area, devono avvalersi sempre di chiamate di sistema per leggere i dati del proprio processo.

Tabella 84.52. Membri del tipo `proc_t`.

Membro	Contenuto
ppid	Numero del processo genitore: <i>parent process id</i> .
pgrp	Numero del gruppo di processi a cui appartiene quello della voce corrispondente: <i>process group</i> . Si tratta del numero del processo a partire dal quale viene definito il gruppo.

Membro	Contenuto
uid	Identità reale del processo della voce corrispondente: <i>user id</i> . Si tratta del numero dell'utente, secondo la classificazione del file <code>/etc/passwd</code> , per conto del quale il processo è stato avviato. Tuttavia, i privilegi del processo dipendono dall'identità efficace, definita dal membro <code>‘euid’</code> .
euid	Identità efficace del processo della voce corrispondente: <i>effective user id</i> . Si tratta del numero dell'utente, secondo la classificazione del file <code>/etc/passwd</code> , per conto del quale il processo è in funzione; pertanto, il processo ha i privilegi di quell'utente.
suid	Identità salvata: <i>saved user id</i> . Si tratta del valore che aveva <i>euid</i> prima di cambiare identità.
gid	Gruppo reale del processo della voce corrispondente: <i>group id</i> . Si tratta del numero del gruppo, secondo la classificazione del file <code>/etc/group</code> , per conto del quale il processo è stato avviato. Tuttavia, i privilegi del processo dipendono dall'identità efficace, definita dal membro <code>‘egid’</code> .
egid	Gruppo efficace del processo della voce corrispondente: <i>effective group id</i> . Si tratta del numero del gruppo, secondo la classificazione del file <code>/etc/group</code> , per conto del quale il processo è in funzione; pertanto, il processo ha i privilegi di quel gruppo.

Membro	Contenuto
sgid	Gruppo salvato: <i>saved group id</i> . Si tratta del valore che aveva <i>egid</i> prima di cambiare identità.
device_tty	Terminale di controllo, espresso attraverso il numero del dispositivo.
path_cwd inode_cwd	Entrambi i membri rappresentano la directory corrente del processo: nel primo caso in forma di percorso, ovvero di stringa, nel secondo in forma di puntatore a inode rappresentato in memoria.
umask	Maschera dei permessi associata al processo: i permessi attivi nella maschera vengono tolti in fase di creazione di un file o di una directory.
sig_status	Segnali inviati al processo e non ancora trattati: ogni segnale si associa a un bit differente del valore del membro <i>sig_status</i> ; un bit a uno indica che il segnale corrispondente è stato ricevuto e non ancora trattato.
sig_ignore	Segnali che il processo ignora: ogni segnale da ignorare si associa a un bit differente del valore del membro <i>sig_ignore</i> ; un bit a uno indica che quel segnale va ignorato.
sig_handler[]	Array di funzioni da eseguire al ricevimento del segnale rispettivo.

Membro	Contenuto
sig_handler_wrapper[]	Array di funzioni da usare per avvolgere quelle da eseguire al ricevimento di un certo segnale. Si tratta in pratica della funzione dichiarata nel file 'lib/signal/_signal_handler_wrapper.s', ma è riferita al codice dell'applicazione di origine. Queste funzioni hanno il compito di sistemare la pila dopo l'esecuzione della funzione attivata da un segnale.
usage	Tempo di utilizzo della CPU, da parte del processo, espresso in impulsi del temporizzatore, il quale li produce alla frequenza di circa 100 Hz.
status	Stato del processo, rappresentabile attraverso una macro-variabile simbolica, definita nel file 'proc.h'. Per os32, gli stati possibili sono: «inesistente», quando si tratta di una voce libera della tabella dei processi; «creato», quando un processo è appena stato creato; «pronto», quando un processo è pronto per essere eseguito, «in esecuzione», quando il processo è in funzione; «sleeping», quando un processo è in attesa di qualche evento; «zombie», quando un processo si è concluso, ha liberato la memoria, ma rimangono le sue tracce perché il genitore non ha ancora recepito la sua fine.

Membro	Contenuto
wakeup_events	Eventi attesi per il risveglio del processo, ammesso che si trovi nello stato si attesa. Ogni tipo di evento che può essere atteso corrisponde a un bit e si rappresenta con una macro-variabile simbolica, dichiarata nel file 'lib/sys/os32.h'.
wakeup_signal	Ammesso che il processo sia in attesa di un segnale, questo membro esprime il numero del segnale atteso.
wakeup_timer	Ammesso che il processo sia in attesa dello scadere di un conto alla rovescia, questo membro esprime il numero di secondi che devono ancora trascorrere.
wakeup_inode	Ammesso che il processo sia in attesa di poter accedere a un inode, questo membro esprime il puntatore a un inode che deve rendersi disponibile.
address_text domain_text	Il valore di questi membri descrive la memoria utilizzata dal processo per le istruzioni (il segmento codice). La voce <i>domain_text</i> rappresenta la dimensione occupata a partire da <i>address_text</i> .
address_data domain_data	Il valore di questi membri descrive la memoria utilizzata dal processo per i dati; tuttavia l'informazione è utile solo se i dati sono distinti dal segmento codice (gli eseguibili di os32 possono essere compilati in modo da condividere codice e dati nello stesso segmento, oppure in modo da tenerli separati).

Membro	Contenuto
<code>domain_stack</code>	Dimensione della memoria usata per la pila dei dati, la quale, a seconda del tipo di eseguibile, può collocarsi nel segmento dati, oppure nell'unico segmento che include codice e dati; in ogni caso, si tratta della porzione finale della memoria in questione.
<code>extra_data</code>	Dimensione della memoria usata per l'allocazione dinamica della memoria. Questo spazio, ammesso che sia utilizzato, si colloca dopo la pila e può essere modificato con la funzione <i>brk()</i> .
<code>sp</code>	Indice della pila dei dati, nell'ambito del segmento dati del processo. Il valore è significativo quando il processo è nello stato di pronto o di attesa di un evento. Quando invece un processo era attivo e viene interrotto, questo valore viene aggiornato.
<code>ret</code>	Rappresenta il valore restituito da un processo terminato e passato nello stato di «zombie».
<code>name[]</code>	Il nome del processo, rappresentato dal nome del programma avviato.
<code>fd[]</code>	Tabella dei descrittori dei file relativi al processo.

L'indice della tabella dei processi corrisponde al numero del processo, ovvero il PID, che infatti non è rappresentato al suo interno. Tuttavia, per accedervi più agevolmente, viene usata la funzione *proc_reference()*, la quale, fornendo il numero PID desiderato, fornisce il puntatore alla voce della tabella che lo descrive(listato

94.14.6).

84.4.3 Chiamate di sistema

«

I processi eseguono una chiamata di sistema attraverso la funzione `sys()`, dichiarata nel file `'lib/sys/os32/sys.s'` (listato [95.21.7](#)). La funzione in sé, per come è dichiarata, potrebbe avere qualunque parametro, ma in pratica ci si attende che il suo prototipo sia il seguente:

```
void sys (int syscallnr, void *message, size_t size);
```

Il numero della chiamata di sistema, richiesto come primo parametro, si rappresenta attraverso una macro-variabile simbolica, definita nel file `'lib/sys/os32.h'`.

Per fornire dei dati a quella parte di codice che deve svolgere il compito richiesto, si usa una variabile strutturata, di cui viene trasmesso il puntatore (riferito al segmento dati del processo che esegue la chiamata) e la dimensione complessiva.

Nel file `'lib/sys/os32.h'` sono definiti dei tipi derivati, riferiti a variabili strutturate, per ogni tipo di chiamata (listato [95.21](#)). Per esempio, per la chiamata di sistema usata per cambiare la directory corrente del processo, si usa un messaggio di tipo `'sysmsg_chdir_t'`:

```
typedef struct {
    char        path[PATH_MAX];
    int         ret;
    int         errno;
    int         errln;
    char        errfn[PATH_MAX];
} sysmsg_chdir_t;
```

In realtà, la funzione `sys()`, si limita a produrre un'interruzione software, da cui viene attivata la routine che inizia al simbolo '`isr_128`' nel file '`kernel/ibm_i386/isr.s`', la quale estrapola le informazioni salienti dalla pila dei dati e poi le fornisce alla funzione `sysroutine()`:

```
void sysroutine (uint32_t syscallnr,
                uint32_t msg_off, uint32_t msg_size);
```

I parametri della funzione `sysroutine()` corrispondono in pratica agli argomenti della chiamata della funzione `sys()`, con la differenza che nei vari passaggi hanno perso l'identità originaria e giungono come numeri puri e semplici. A questo proposito, il secondo parametro cambia nome, in quanto ciò che prima era il puntatore a un'area di memoria, qui va interpretato come lo scostamento rispetto al segmento dati del processo (*segment offset*).

84.4.4 Funzione «`proc_init()`»

«

Listato [94.14.3](#).

```
void proc_init (void);
```

La funzione *proc_init()* viene chiamata dalla funzione *kmain()*, una volta sola, per attivare la gestione dei processi elaborativi. Si occupa di compiere le azioni seguenti:

- predisporre la tabella GDT, attraverso la chiamata della funzione *gdt()*;
- impostare il temporizzatore in modo da fornire impulsi alla frequenza dichiarata nella macro-variabile *CLOCKS_PER_SEC*, pari a 100 Hz;
- predisporre la tabella IDT, attraverso la chiamata della funzione *idt()*;
- azzerare la tabella dei processi, inserendovi però i dati relativi al kernel (il processo zero);
- allocare la memoria già utilizzata dal kernel;
- attivare selettivamente le interruzioni hardware desiderate;
- attivare la gestione delle unità PATA;
- innestare il file system principale.

84.4.5 Funzione «sysroutine()»

Listato [94.14.28](#).

La funzione *sysroutine()* viene chiamata esclusivamente dalla routine attivata dalle chiamate di sistema (tale routine è introdotta dal simbolo '*isr_128*' nel file '*kernel/ibm_i386/isr.s*') e ha i parametri che si possono vedere dal prototipo:



```
void sysroutine (uint32_t syscallnr,
                uint32_t msg_off, uint32_t msg_size);
```

Il primo parametro è il numero della chiamata di sistema che ha provocato l'interruzione; gli altri due danno la posizione e la dimensione del messaggio inviato attraverso la chiamata di sistema.

All'inizio della funzione viene dichiarato un puntatore a un'unione di tutti i tipi di messaggio gestibili:

```
union {
    sysmsg_brk_t      brk;
    sysmsg_chdir_t   chdir;
    sysmsg_chmod_t   chmod;
    ...
} *msg;
```

Viene quindi calcolata la collocazione del messaggio originale, per poi poter assegnare a *msg* il puntatore a tale messaggio.

Le chiamate di sistema sono fatte per le applicazioni, ma al kernel è consentito di eseguirne alcune, per motivi particolari. Se però il kernel tenta di eseguire una chiamata differente, si ottiene un messaggio di avvertimento, ma si tenta ugualmente l'esecuzione della richiesta.

Disponendo del puntatore *msg*, sapendo di quale chiamata di sistema si tratta, il messaggio può essere letto come:

```
msg->tipo_chiamata
```

Per esempio, per la chiamata di sistema '**SYS_CHDIR**', si deve fare riferimento al messaggio *msg->chdir*; pertanto, per raggiungere il

membro *ret* del messaggio si usa la notazione *msg->chdir.ret*.

Per distinguere il tipo di chiamata si usa una struttura di selezione:

```
switch (syscallnr)
{
    case SYS_0:
        break;
    case SYS_BRK:
        msg->brk.ret = s_brk (pid, msg->brk.address);
        sysroutine_error_back (&msg->brk.errno,
                               &msg->brk.errln,
                               msg->brk.errfn);

        break;
    case SYS_CHDIR:
        msg->chdir.ret = s_chdir (pid, msg->chdir.path);
        sysroutine_error_back (&msg->chdir.errno,
                               &msg->chdir.errln,
                               msg->chdir.errfn);

        break;
```

Il messaggio usato per trasmettere i dati della chiamata, può servire anche per restituire dei dati al mittente, pertanto, spesso alcuni contenuti vengono modificati. Ciò succede particolarmente con il membro *ret* che generalmente rappresenta il valore restituito dalla chiamata di sistema.

Al termine del lavoro, viene chiamata la funzione *proc_scheduler()*.

84.4.6 Funzione «proc_scheduler()»

Listato [94.14.11](#).

La funzione *proc_scheduler()* non prevede parametri e riceve le informazioni che le possono servire attraverso variabili pubbliche: *_ksp*, *proc_stack_pointer*, *proc_stack_segment_selector* e

proc_current. A sua volta, la funzione aggiorna i valori di queste variabili, per mettere in pratica uno scambio di processi.

```
void proc_scheduler (void);
```

La prima cosa che fa la funzione consiste nel verificare che il valore dell'indice della pila del processo interrotto non superi lo spazio disponibile per la pila stessa. Diversamente il processo viene eliminato forzatamente, con una segnalazione adeguata sul terminale attivo. Si ottiene comunque una segnalazione se l'indice si avvicina pericolosamente al limite.

Successivamente la funzione svolge delle operazioni che riguardano tutti i processi: aggiorna i contatori dei processi che attendono lo scadere di un certo tempo; verifica la presenza di segnali e predispone le azioni relative; raccoglie l'input dai terminali.

```
proc_sch_timers ();  
...  
proc_sch_signals ();  
...  
proc_sch_terminals ();
```

A quel punto aggiorna il tempo di utilizzo della CPU del processo appena interrotto:

```
current_clock = s_clock ((pid_t) 0);  
proc_table[prev].usage += current_clock - previous_clock;  
previous_clock = current_clock;
```

Quindi inizia la ricerca di un altro processo, candidato a essere ripreso al posto di quello interrotto. La ricerca inizia dal processo successivo a quello interrotto, senza considerare alcun criterio di prece-

denza. Il ciclo termina se la ricerca incontra di nuovo il processo di partenza.

```
for (next = prev+1; next != prev; next++)
{
    if (next >= PROCESS_MAX)
    {
        next = -1; // At the next loop, 'next'
                  // will be zero.
        continue;
    }
    ...
}
```

All'interno di questo ciclo di ricerca, se si incontra un processo pronto per essere messo in funzione, lo si scambia con quello interrotto: in pratica si salva il valore attuale dell'indice della pila, si scambiano gli stati e si aggiornano i valori di *proc_current*, *proc_stack_segment_selector* e *proc_stack_pointer*, in modo da ottenere effettivamente lo scambio all'uscita dalla funzione:

```
else if (proc_table[next].status == PROC_READY)
{
    if (proc_table[prev].status == PROC_RUNNING)
    {
        proc_table[prev].status = PROC_READY;
    }
    proc_table[prev].sp      = proc_stack_pointer;
    proc_table[next].status = PROC_RUNNING;
    proc_table[next].ret     = 0;
    proc_current             = next;
    proc_stack_segment_selector
        = gdt_pid_to_segment_data (next) * 8;
    proc_stack_pointer      = proc_table[next].sp;
    break;
}
```

Alla fine del ciclo, occorre verificare se esiste effettivamente un processo successivo attivato, perché in caso contrario, si lascia il controllo direttamente al kernel. In fine, si salva il valore accumulato in precedenza dell'indice della pila del kernel, nella variabile *_ksp*.

84.4.7 Programmazione dei segnali

«

Un processo può ricevere un segnale, a seguito del quale può essere interrotto per compiere una certa azione. La maggior parte dei segnali può essere inibita, in modo tale che ricevendoli il processo non venga a essere disturbato, oppure si può associare loro una funzione, da eseguire al momento del ricevimento del tale segnale. Diversamente, in mancanza di tale associazione, il ricevimento di un segnale comporta un'azione predefinita.

L'associazione di una funzione allo scattare di un segnale si ottiene, nel codice dell'applicazione, con la funzione *signal()* (lista-

to [95.17.3](#)), la quale attraverso una chiamata di sistema fornisce al kernel tutti i dati necessari per la programmazione del segnale.

La vera difficoltà sta nell'esecuzione effettiva della funzione, nel momento in cui scatta il segnale previsto per il processo.

```
sighandler_t signal (int sig, sighandler_t handler);
```

La funzione *signal()* richiede l'indicazione del numero del segnale da programmare e di un puntatore rappresentato da una funzione che si vuole azionare nel momento in cui scatta il segnale in questione. Il tipo '**sighandler_t**' rappresenta il puntatore a una funzione che richiede un parametro di tipo intero, costituito dal numero del segnale ricevuto, e non restituisce alcunché; pertanto, la funzione che si passa come secondo parametro della funzione *signal()* deve avere la forma seguente:

```
void handler (int sig);
```

La funzione *signal()*, a sua volta, esegue finalmente la chiamata di sistema, ma oltre al numero del segnale e al puntatore della funzione da azionare, invia il puntatore di un'altra funzione, denominata *_sighandler_wrapper()*, il cui scopo è quello di avvolgere la chiamata della funzione da azionare, per sistemare in modo appropriato la pila dei dati (listato [95.17.1](#)). In questa fase della descrizione del problema, va osservato che la funzione *_sighandler_wrapper()* si trova nel codice del processo che riceve il segnale.

La chiamata di sistema, quando raggiunge il kernel, comporta l'aggiornamento dei dati del processo, annotando sia la funzione da

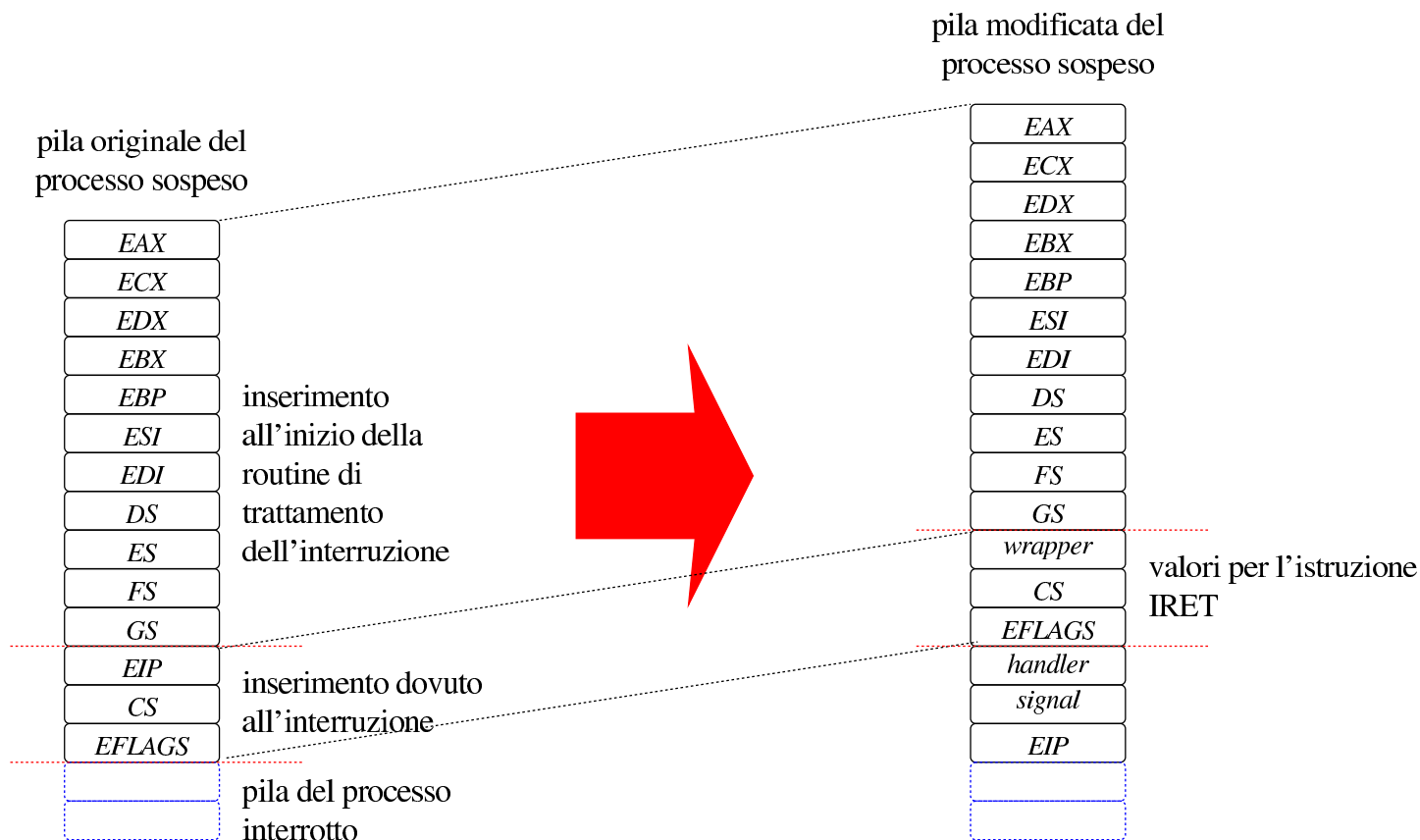
azionare, sia la funzione che deve avvolgerla.

Quando arriva un segnale a un processo che prevede l'azionamento di una funzione, attraverso la funzione *proc_sch_signals()*, chiamata a sua volta dalla funzione *proc_scheduler()*, attraverso altri passaggi si arriva alla funzione *proc_sig_handler()* (listato [94.14.15](#)).

```
void proc_sig_handler (pid_t pid, int sig);
```

La funzione *proc_sig_handler()* ha lo scopo di modificare la pila dei dati del processo *pid*, in modo da far sì che, nel momento in cui fosse selezionato, prima di riprendere con l'attività sospesa originariamente, esegua la funzione attivata dal segnale *sig*.

Figura 84.60. Modifica della pila attraverso la funzione *proc_sig_handler()*.



Come si può vedere nella figura, i valori che servono all'istruzione IRET per concludere l'interruzione, vengono modificati in modo da ripartire iniziando con la funzione che avvolge quella da azionare, *wrapper*, ovvero quella che dal lato dell'applicazione è chiamata *_sighandler_wrapper()*. D'altro canto, quando quella funzione viene messa in azione, si trova nella pila dei valori che le servono per poter chiamare a sua volta la funzione da azionare effettivamente.

Il codice di *_sighandler_wrapper()* non corrisponde propriamente a una funzione, in quanto ciò che si trova nella pila non è quello che si prevede di solito. Le figure successive mostrano i cambiamenti della pila del processo, prima e dopo l'esecuzione della funzione incaricata di gestire il segnale ricevuto.

Figura 84.61. Dall'avvio di `_sighandler_wrapper()` fino alla chiamata della funzione di gestione del segnale ricevuto.

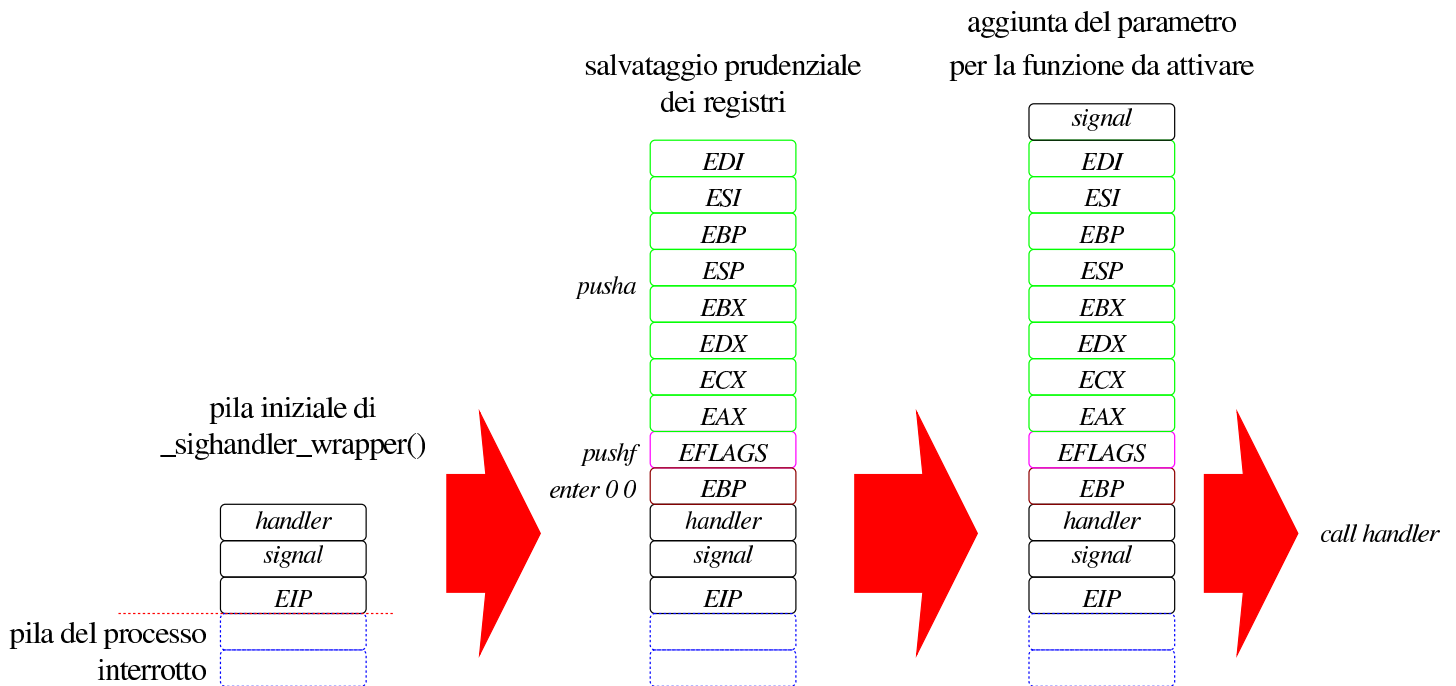
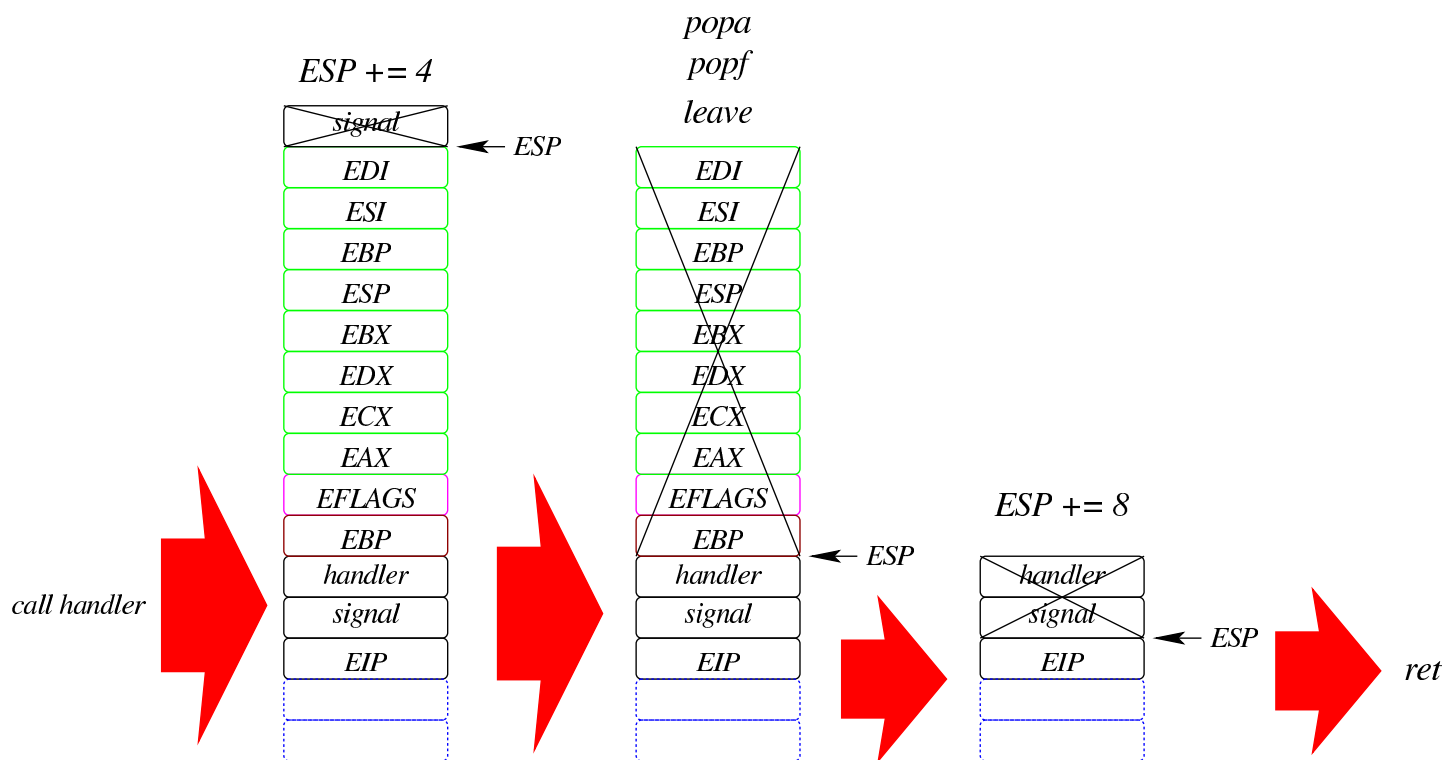


Figura 84.62. Dopo la conclusione della funzione di gestione del segnale ricevuto, fino alla restituzione del controllo al termine di `_sighandler_wrapper()`.



Lo scopo della funzione `_sighandler_wrapper()` è quello di garantire che sia preservato completamente l'ambiente di lavoro del processo nel momento dell'interruzione, perché non è possibile fare affidamento sul rispetto delle convenzioni di chiamata, dato che la funzione da azionare in corrispondenza dell'interruzione, viene iniettata in una posizione arbitraria del codice.

Riquadro 84.63. Programmazione ripetuta dei segnali.

Il sistema tradizionale con cui si programma una funzione in corrispondenza di un segnale, richiede che lo scattare del segnale riporti la gestione di questo allo stato predefinito, perché altrimenti la stessa funzione azionata potrebbe essere interrotta da un segnale che ne aziona un'altra. Se così fosse, la pila dei dati potrebbe riempirsi velocemente, portando il processo al collasso. Di conseguenza, se il programmatore desidera ripristinare una funzione associata a un segnale, lo deve richiedere alla fine della funzione stessa (chiamando *signal()* di nuovo), ma in tal caso c'è la possibilità che quel segnale raggiunga il processo nell'intervallo di tempo tra l'azionamento della funzione e il ripristino della programmazione della stessa. Per esempio, ciò significa che se si vuole controllare il segnale *SIG_TERM* per impedire che questo porti alla conclusione il processo, anche se la funzione azionata dal segnale richiama *signal()* per programmarsì nuovamente dopo la chiamata, in modo da non perdere il controllo del segnale, se il processo viene interessato da una raffica di segnali *SIG_TERM*, prima o poi il processo viene concluso, perché un segnale lo raggiunge quando quella funzione non ha ancora fatto in tempo a chiamare *signal()*.

84.4.8 Salvataggio e recupero della pila per i «salti non locali»

«

Il linguaggio C prevede la disponibilità di due funzioni, attraverso le quali è possibile salvare il contesto della pila dei dati per poterne recuperare lo stato in un momento successivo. Tuttavia, tale recupero può avvenire solo se dopo il salvataggio il contenuto della pila precedente rimane valido, in quanto il recupero avviene solo in una situazione in cui la pila sia stata incrementata ulteriormente.

Il salvataggio si ottiene con la funzione *setjmp()* e il recupero con

longjmp(). L'effetto della chiamata della funzione *longjmp()* comporta il riportare il processo alla situazione in cui si trovava dopo l'esecuzione della funzione *setjmp()*, con la differenza che nel secondo caso, la funzione *setjmp()* restituisce un valore differente.

Si tratta di un modo pessimo di programmare, tuttavia fa parte dello standard del linguaggio C.

Generalmente, la realizzazione delle funzioni *setjmp()* e *longjmp()* avviene nella libreria, senza coinvolgere il kernel in alcun modo. Ma os32 procede diversamente e si avvale invece di chiamate di sistema. Si tratta comunque di una scelta motivata esclusivamente da una più semplice comprensione del codice, facendo rientrare il meccanismo in quello più generale della gestione dei processi.

La funzione *setjmp()* è realizzata dal file 'lib/setjmp/setjmp.s' (listato 95.16.2). La funzione svolge sostanzialmente il compito che si può vedere tradotto in linguaggio C nel codice seguente, se il compilatore gestisse la pila dei dati nella forma più compatta e prevedibile:

```
#include <sys/os32.h>
#include <setjmp.h>
int
setjmp (jmp_buf env)
{
    sysmsg_jump_t msg;
    msg.env = env;
    msg.ret = 0;
    sys (SYS_SETJMP, &msg, sizeof msg);
    return (msg.ret);
}
```

La funzione *longjmp()* è realizzata invece in C, nel file 'lib/

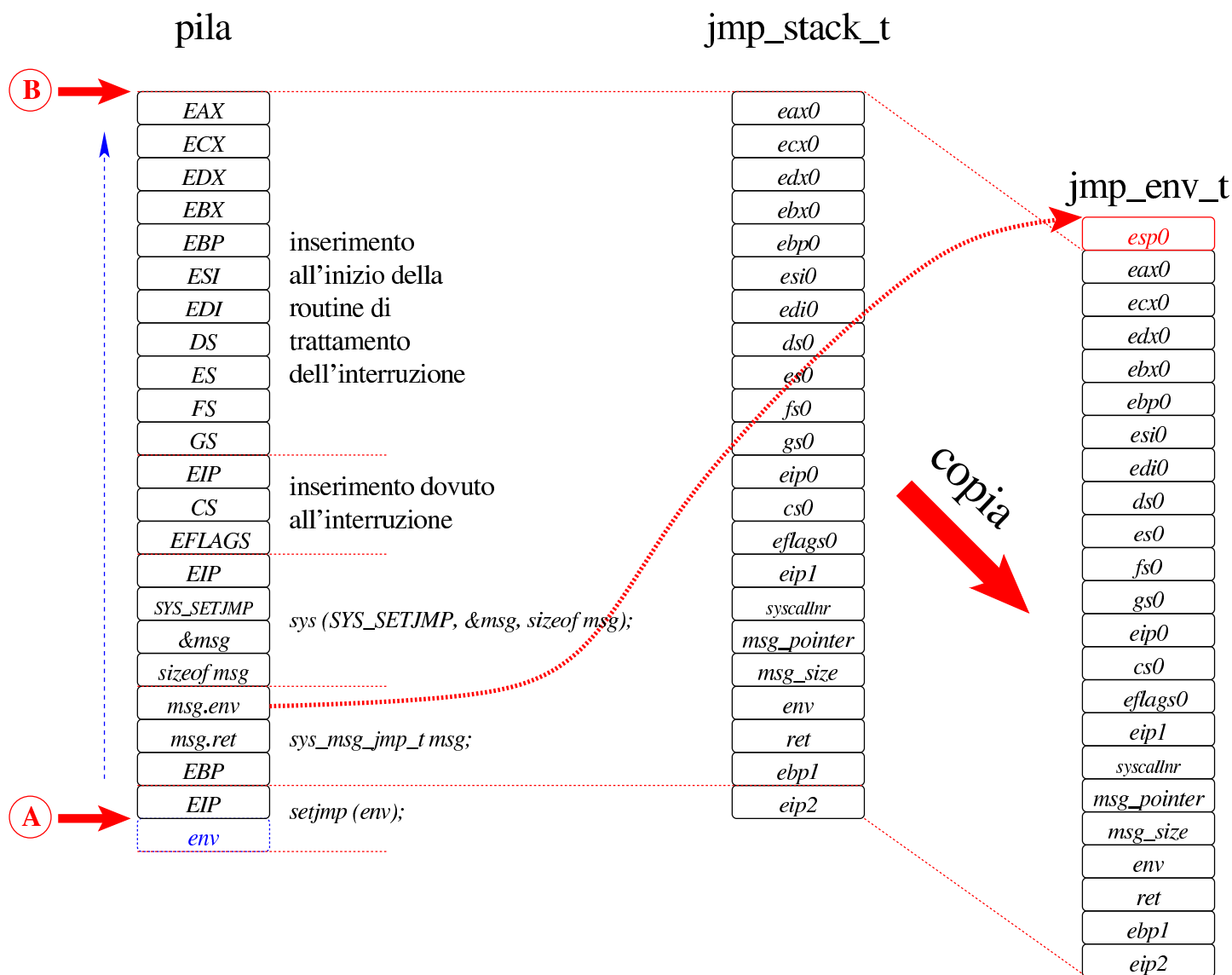
`setjmp/longjmp.c` (listato [95.16.1](#)), perché non c'è la necessità di conoscere esattamente la struttura della sua pila.

La struttura corrispondente al tipo '`sysmsg_jump_t`' si limita a due campi: un puntatore che deve fare riferimento alla memoria in cui viene salvato il contenuto della pila e il valore che deve restituire `setjmp()` quando rivive attraverso la chiamata di `longjmp()`.

```
typedef struct {
    void *env;
    int   ret;
} sysmsg_jump_t;
```

Le due chiamate di sistema raggiungono, rispettivamente, le funzioni `s_setjmp()` e `s_longjmp()` del kernel (listati [94.8.38](#) e [94.8.22](#)). La funzione `s_setjmp()` salva lo stato della pila, a partire dalla chiamata della funzione `setjmp()`, mentre `s_longjmp()` lo ripristina, rimettendo anche l'indice della pila allo stato che aveva al momento della chiamata di `setjmp()`.

Figura 84.66. Lo stato della pila durante le varie fasi che riguardano la chiamata di *setjmp()*, a confronto con i tipi 'jmp_stack_t' e 'jmp_env_t'.



La funzione *setjmp()* prevede un argomento di tipo 'jmp_buf' che lo standard prescrive sia come un array:

```
int setjmp (jmp_buf env);
```

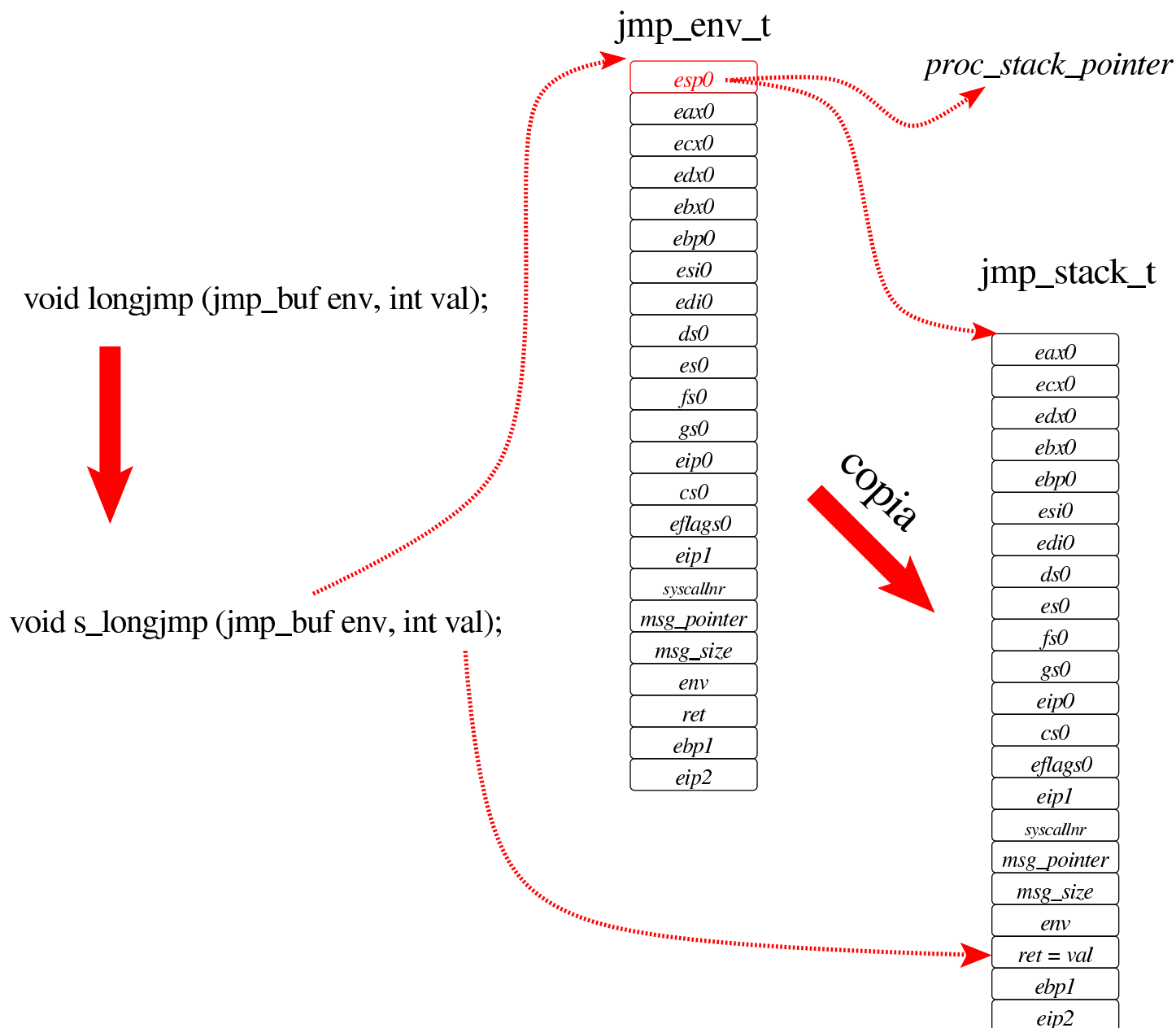
In pratica, l'array serve solo a occupare lo spazio necessario a rappresentare il tipo 'jmp_env_t', i cui membri si vedono rappresentati

nella figura già apparsa. La funzione *s_setjmp()* si occupa di salvare lo stato della pila, dal punto «A» al punto «B» della figura, all'interno di *env*, secondo la struttura di '*jmp_env_t*', mettendo, oltre al contenuto della pila, il valore del suo indice attuale.

La funzione *longjmp()* deve portare al ripristino della pila, in una posizione antecedente rispetto a quella attuale.

```
void longjmp (jmp_buf env, int val);
```

Figura 84.67. La chiamata di *longjmp()* ricostruisce la vecchia pila di *setjmp()*, nella posizione in cui si trovava, ricollocando l'indice della pila e modificando il valore che poi *setjmp()* rediviva va a restituire.

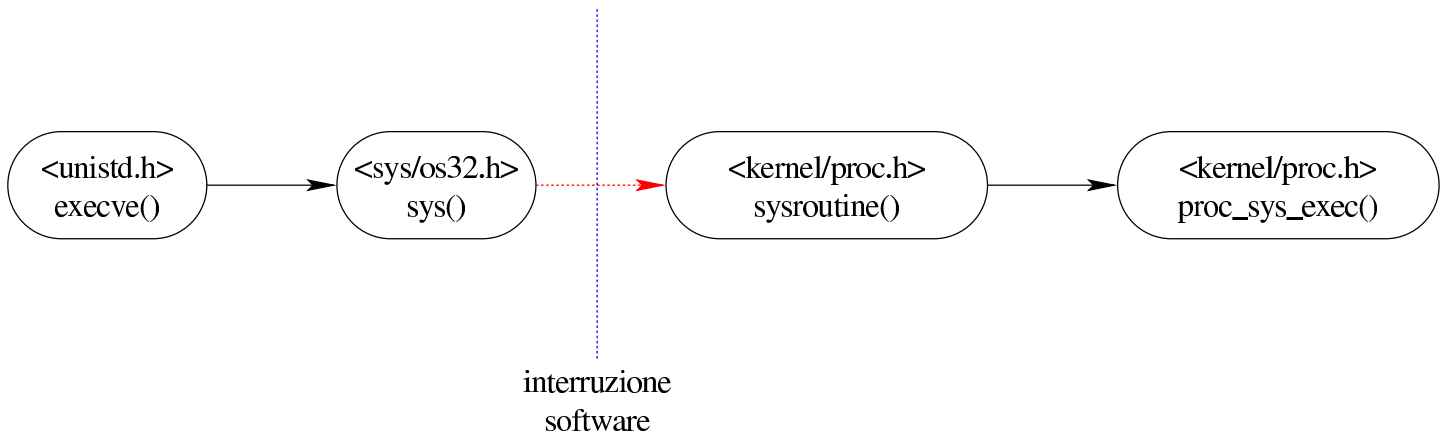


84.5 Caricamento ed esecuzione delle applicazioni

«

Caricare un programma e metterlo in esecuzione è un processo delicato che parte dalla funzione *execve()* della libreria standard e viene svolto dalla funzione *proc_sys_exec()* del kernel.

Figura 84.68. Da *execve()* a *proc_sys_exec()*.



84.5.1 Caricamento in memoria

«

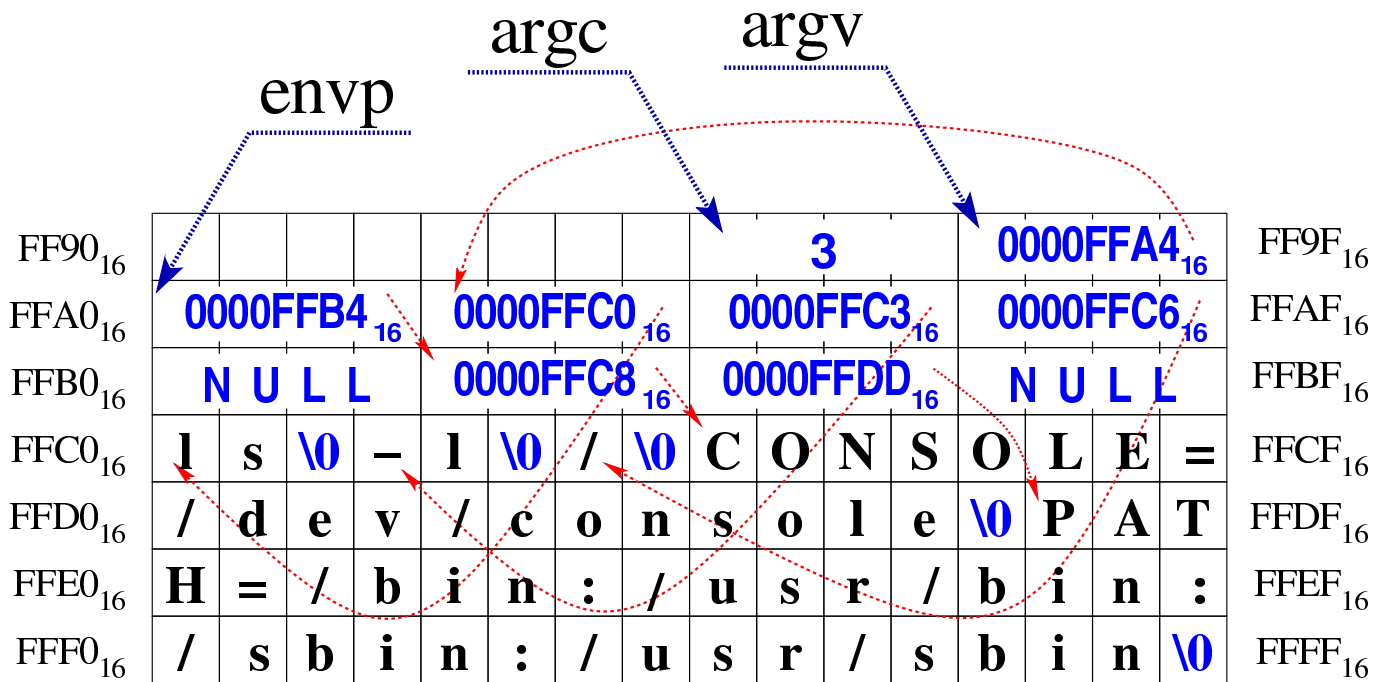
La funzione *proc_sys_exec()* (listato [94.14.22](#)) del kernel è quella che svolge il compito di caricare un processo in memoria e di annotarlo nella tabella dei processi.

La funzione, dopo aver verificato che si tratti di un file eseguibile valido e che ci siano i permessi per metterlo in funzione, procede all'allocazione della memoria, dividendo se necessario l'area codice da quella dei dati, quindi legge il file e copia opportunamente le componenti di questo nelle aree di memoria allocate.

La realizzazione attuale della funzione *proc_sys_exec()* non è in grado di verificare se un processo uguale sia già in memoria, quindi carica la parte del codice anche se questa potrebbe essere già disponibile.

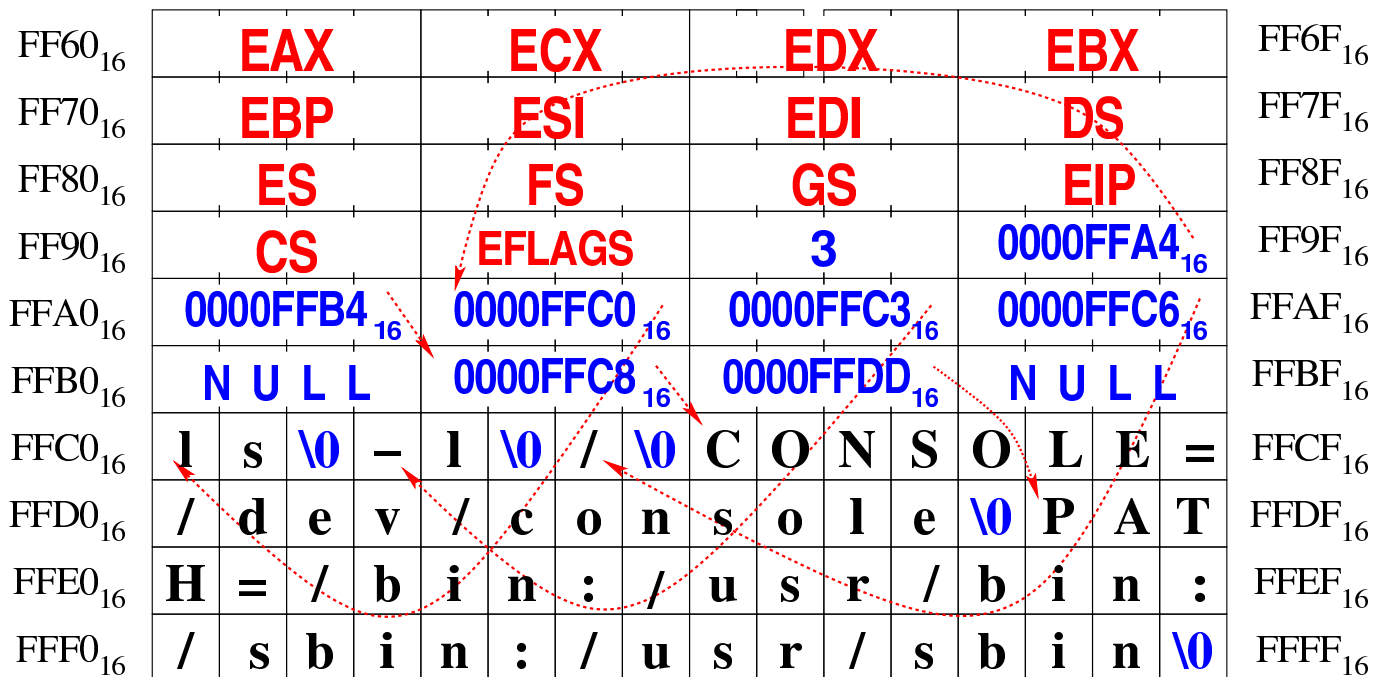
Terminato il caricamento del file viene aggiornata la tabella GDT e quindi viene ricostruita in memoria la pila dei dati del processo. Prima si mettono sul fondo le stringhe delle variabili di ambiente e quelle degli argomenti della chiamata, quindi si aggiungono i puntatori alle stringhe delle variabili di ambiente, ricostruendo così l'array noto convenzionalmente come '*envp[]*', continuando con l'aggiunta dei puntatori alle stringhe degli argomenti della chiamata, per riprodurre l'array '*argv[]*'. Per ricostruire gli argomenti della chiamata della funzione *main()* dell'applicazione, vanno però aggiunti ancora: il puntatore all'inizio dell'array delle stringhe che descrivono le variabili di ambiente, il puntatore all'array delle stringhe che descrivono gli argomenti della chiamata e il valore che rappresenta la quantità di argomenti della chiamata.

Figura 84.69. Caricamento degli argomenti della chiamata della funzione *main()*.



Fatto ciò, vanno aggiunti tutti i valori necessari allo scambio dei processi, costituiti dai vari registri da rimpiazzare.

Figura 84.70. Completamento della pila con i valori dei registri.



Superato il problema della ricostruzione della pila dei dati, la funzio-

ne *proc_sys_exec()* predispone i descrittori di standard input, standard output e standard error, quindi libera la memoria usata dal processo chiamante e ne rimpiazza i dati nella tabella dei processi con quelli del nuovo processo caricato.

84.5.2 Il codice iniziale dell'applicativo

I programmi iniziano con il codice che si trova nel file 'applic/crt0.mer.s', oppure 'applic/crt0.sep.s', a seconda che si compilino in modo da avere codice e dati nello stesso segmento, oppure in segmenti di memoria differenti. Questo file è abbastanza diverso da 'kernel/main/crt0.s' del kernel; in particolare va osservato che, a differenza del kernel, il codice delle applicazioni viene eseguito in un momento in cui l'indice della pila è già collocato correttamente; inoltre, se la funzione *main()* delle applicazioni termina e restituisce il controllo a 'crt0.*.s', un ciclo senza fine esegue continuamente una chiamata di sistema per la conclusione del processo elaborativo corrispondente.

Figura 84.71. Codice iniziale degli applicativi e variabile strutturata di tipo ‘**header_t**’.

```

.section .text
startup:
    jmp startup_code
filler:
    .space (0x0004 - (filler - startup))
magic:
    .quad 0x6F7333326170706C    # os32appl
doffset:
    .int _text_start | _text_end
etext:
    .int _text_end
edata:
    .int _data_end
ebss:
    .int _bss_end
stack_size:
    .int 0x8000
.align 4
startup_code:
...

```

```

typedef struct {
    uint32_t filler0;
    uint64_t magic;
    uint32_t data_offset;
    uint32_t etext;
    uint32_t edata;
    uint32_t ebss;
    uint32_t ssize;
} header_t;

```

La figura mostra il confronto tra il codice iniziale contenuto nel file ‘`applic/crt0.*.s`’, senza preamboli e senza commenti, con la dichiarazione del tipo derivato ‘**header_t**’, presente nel file ‘`kernel/proc.h`’ (nel codice si può notare la differenza tra ‘`crt0.mer.s`’ e ‘`crt0.sep.s`’, relativa al valore assegnato alla variabile **doffset**). Attraverso questa struttura, la funzione **proc_sys_exec()** è in grado di estrapolare dal file le informazioni necessarie a caricarlo correttamente in memoria.

Come già accennato, quando viene eseguito il codice di un programma applicativo, la pila dei dati è già operativa. Pertanto, dopo il

simbolo `'startup_code'` si può già lavorare con questa.

```
pop %eax           # argc
pop %ebx           # argv
pop %ecx           # envp
mov %ecx, environ # Variable 'environ' comes
                  # from <unistd.h>.

push %ecx
push %ebx
push %eax
```

Per prima cosa, viene estratto dalla pila il puntatore all'array noto come *envp[]*, per poter assegnare tale valore alla variabile *environ*, come richiede lo standard della libreria POSIX. Tuttavia, per poter gestire poi le variabili di ambiente, si rende necessario utilizzare un array più «comodo», quando le stringhe vanno sostituite. A tale proposito, nel file `'lib/stdlib/environment.c'`, si dichiarano *_environment_table[][]* e *_environment[]*. Il primo è semplicemente un array di caratteri, dove, utilizzando due indici di accesso, si conviene di allocare delle stringhe, con una dimensione massima prestabilita. Il secondo, invece, è un array di puntatori, per localizzare l'inizio delle stringhe contenute nel primo. In pratica, alla fine *_environment[]* e *environ[]* devono essere equivalenti. Ma per attuare questo, occorre utilizzare la funzione *_environment_setup()* che sistema tutti i puntatori necessari.

```
push %ecx
call _environment_setup
add $4, %esp

mov $_environment, %eax
mov %eax, environ

pop %eax          # argc
pop %ebx          # argv[][]
pop %ecx          # envp[][]
mov $_environment, %ecx
push %ecx
push %ebx
push %eax
```

Come si vede dall'estratto del file 'applic/crt0.*.s', si vede l'uso della funzione `_environment_setup()` (il registro ECX contiene già il puntatore a `envp[]`, e viene inserito nella pila proprio come argomento per la funzione). Successivamente viene riassegnata anche la variabile `environ` in modo da coincidere con `_environment`. Alla fine, viene ricostruita la pila per gli argomenti della chiamata della funzione `main()`, ma prima di procedere con quella chiamata, si utilizzano delle funzioni, per inizializzare la gestione dei flussi di file e delle directory, sempre in forma di flussi, e per predisporre la tabella delle funzioni da eseguire alla conclusione del processo.

```
    call _stdio_stream_setup
    call _dirent_directory_stream_setup
    call _atexit_setup

    call main

    mov  %eax, exit_value
    ...
.align 4
.section .data
exit_value:
    .int 0x00000000
.align 4
.section .bss
```

La funzione `_stdio_stream_setup()`, contenuta nel file `'lib/stdio/FILE.c'`, associa i descrittori standard ai flussi di file standard (standard input, standard output e standard error); la funzione `_dirent_directory_stream_setup()`, contenuta nel file `'lib/dirent/DIR.c'`, compie un lavoro analogo, limitandosi però a inizializzare un array di flussi di directory; la funzione `_atexit_setup()`, contenuta nel file `'lib/stdlib/atexit.c'` azzererà l'array `_atexit_table[]`, destinato a contenere l'elenco di funzioni da eseguire alla conclusione del processo.

Dopo queste preparazioni, viene chiamata la funzione `main()`, la quale riceve regolarmente i propri argomenti previsti. Il valore restituito dalla funzione viene poi salvato in corrispondenza del simbolo `'exit_value'`.

```
halt:
    pushl $2           # Size of message.
    pushl $exit_value # Pointer to the message.
    pushl $6          # SYS_EXIT
    call sys
    add $4, %esp
    add $4, %esp
    add $4, %esp
    jmp halt
```

All'uscita dalla funzione *main()*, dopo aver salvato quanto restituito dalla funzione stessa, ci si introduce nel codice successivo al simbolo **'halt'**, nel quale si chiama la funzione *sys()* (chiamata di sistema), per produrre la chiusura formale del processo. Ciò che si vede è comunque l'equivalente di **'_exit (exit_value) ;'**.

84.6 Gestione della memoria



Dal punto di vista del kernel di os32, l'allocazione della memoria riguarda la collocazione dei processi elaborativi nella stessa. Per semplicità si utilizza una mappa di bit per indicare lo stato dei blocchi di memoria, dove un bit a uno indica un blocco di memoria occupato.

Nel file **'memory.h'** viene definita la dimensione di un blocco di memoria e, di conseguenza, la quantità massima che possa essere gestita. Attualmente i blocchi sono da 4096 byte, pertanto, sapendo che la memoria può arrivare solo fino a 4 Gbyte, si gestiscono al massimo 1048576 blocchi.

Per la scansione della mappa si utilizzano interi da 32 bit, pertanto tutta la mappa si riduce a 32768 di questi interi, ovvero 128 Kibyte. Nell'ambito di ogni intero da 32 bit, il bit più significativo rappresenta il primo blocco di memoria di sua competenza. Per esempio, per

indicare che si stanno utilizzando i primi 28672 byte, pari ai primi 7 blocchi di memoria, si rappresenta la mappa della memoria come «FE000000...».

Il fatto che la mappa della memoria vada scandito a ranghi di 32 bit va tenuto in considerazione, perché se invece si andasse con ranghi differenti, si incapperebbe nel problema dell'inversione dei byte.

84.6.1 File «kernel/memory.h» e «kernel/memory/...»

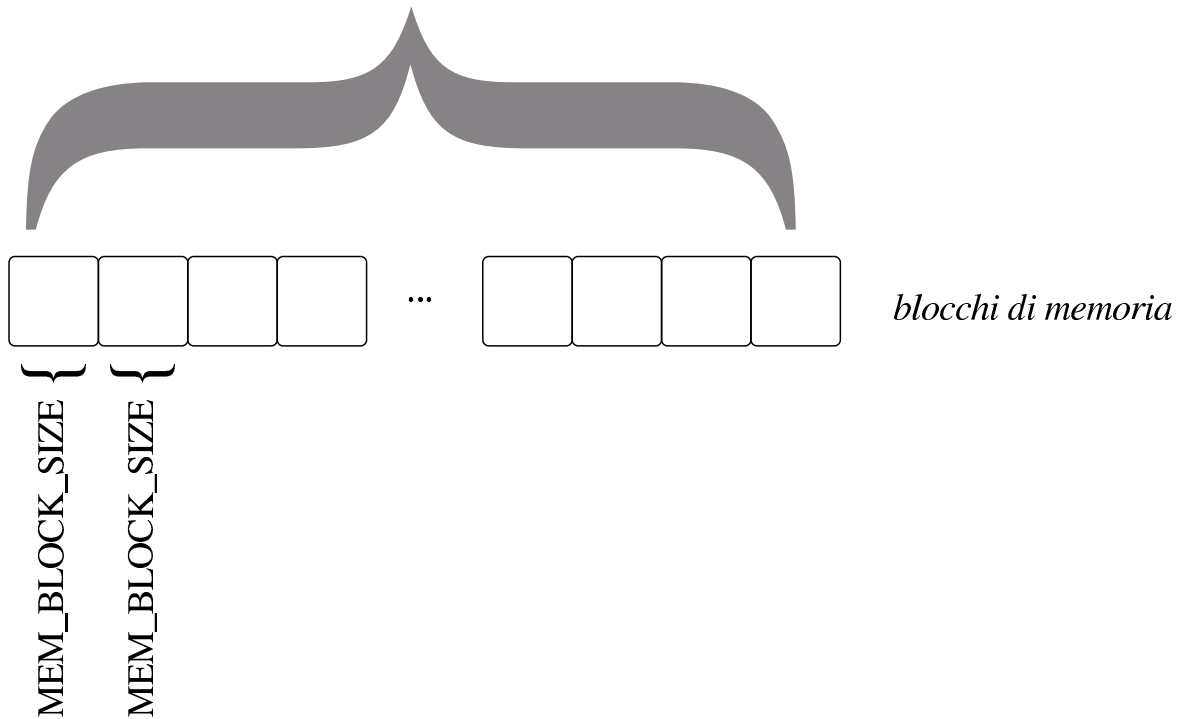
Listato [94.10](#) e successivi.

Il file 'kernel/memory.h', oltre ai prototipi delle funzioni usate per la gestione della memoria, definisce la dimensione del blocco minimo di memoria e la quantità massima di questi, rispettivamente con le macro-variabili *MEM_BLOCK_SIZE* e *MEM_MAX_BLOCKS*; inoltre predispone il tipo derivato 'addr_t', corrispondente a un indirizzo di memoria reale.



Figura 84.76. Mappa della memoria in blocchi: la dimensione minima di un'area di memoria è di ***MEM_BLOCK_SIZE*** byte.

$4294967296 \text{ byte} == 4 \text{ Mibyte} == \text{MEM_MAX_BLOCKS} * \text{MEM_BLOCK_SIZE}$



Nei file della directory `'kernel/memory/'` viene dichiarata la mappa della memoria, corrispondente a un array di interi a 32 bit, denominato ***mb_table[]***. L'array è pubblico, tuttavia è disponibile anche una funzione che ne restituisce il puntatore: ***mb_reference()***. Tale funzione sarebbe perfettamente inutile, ma rimane per uniformità rispetto alla gestione delle altre tabelle.

Tabella 84.77. Funzioni per la gestione della mappa della memoria, dichiarate nel file di intestazione ‘kernel/memory.h’ e realizzate nella directory ‘kernel/memory/’.

Funzione	Descrizione
<pre>uint32_t *mb_reference (void);</pre>	<p>Restituisce il puntatore alla tabella dei blocchi di memoria, per uniformare l'accesso alla tabella dalle funzioni che non fanno parte del gruppo contenuto nella directory ‘kernel/memory/’.</p>
<pre>ssize_t mb_alloc (addr_t <i>address</i>, size_t <i>size</i>);</pre>	<p>Alloca la memoria a partire dall'indirizzo indicato, per la quantità di byte richiesta. L'allocazione ha termine anticipatamente se si incontra un blocco già utilizzato. La funzione restituisce la dimensione allocata effettivamente.</p>
<pre>void mb_free (addr_t <i>address</i>, size_t <i>size</i>);</pre>	<p>Libera la memoria a partire dall'indirizzo indicato, per la quantità di byte richiesta. Lo spazio viene liberato in ogni caso, anche se risulta già libero; tuttavia viene prodotto un avvertimento a video se si verifica tale ipotesi.</p>

Funzione	Descrizione
<pre>int mb_reduce (addr_t <i>address</i>, size_t <i>new</i>, size_t <i>previous</i>);</pre>	Riduce un'area di memoria già utilizzata. Restituisce zero se l'operazione si conclude con successo, oppure -1 in caso contrario, aggiornando la variabile <i>errno</i> di conseguenza.
<pre>void mb_clean (addr_t <i>address</i>, size_t <i>size</i>);</pre>	Azzera l'area di memoria specificata.
<pre>addr_t mb_alloc_size (size_t <i>size</i>);</pre>	Alloca un'area di memoria della dimensione richiesta, restituendone l'indirizzo. La funzione conclude con successo il proprio lavoro se il valore restituito è diverso da zero; se invece l'indirizzo ottenuto è pari a zero si è verificato un errore che può essere verificato analizzando il contenuto della variabile <i>errno</i> .

Funzione	Descrizione
<pre>void mb_size (size_t <i>size</i>);</pre>	<p>Questa funzione, usata una sola volta all'interno di <i>kmain()</i>, serve a definire la dimensione massima della memoria disponibile in blocchi. In pratica, le si fornisce la dimensione effettiva della memoria che viene così divisa per la dimensione del blocco, ignorando il resto. Questa informazione viene conservata nella variabile <i>mb_max</i>.</p>
<pre>void mb_print (void);</pre>	<p>Funzione diagnostica che visualizza gli intervalli di memoria utilizzati, esprimendoli però in blocchi.</p>

84.6.2 Scansione della mappa di memoria

Listato [94.10](#) e successivi.

La mappa della memoria si rappresenta (a sua volta in memoria), con un array di interi a 32 bit, dove ogni bit individua un blocco di memoria. Pertanto, l'array si compone di una quantità di elementi pari al valore di *MEM_MAX_BLOCKS* diviso 32.

Il primo elemento di questo array, ovvero *mb_table[0]*, individua i primi 32 blocchi di memoria, dove il bit più significativo si riferisce precisamente al primo blocco. Per esempio, se *mb_table[0]* contie-



ne il valore $F8000000_{16}$, ovvero 1111100000000000_2 , significa che i primi cinque blocchi di memoria sono occupati, mentre i blocchi dal sesto al trentaduesimo sono liberi.

Dal momento che i calcoli per individuare i blocchi di memoria e per intervenire nella mappa relativa, possono creare confusione, queste operazioni sono raccolte in funzioni statiche separate, anche se sono utili esclusivamente all'interno del file in cui si trovano. Tali funzioni statiche hanno una sintassi comune:

```
int mb_block_set1    (int block)
int mb_block_set0    (int block)
int mb_block_status (int block)
```

Le funzioni *mb_block_set1()* e *mb_block_set0()* servono rispettivamente a impegnare o liberare un certo blocco di memoria, individuato dal valore dell'argomento. La funzione *mb_block_status()* restituisce uno nel caso il blocco indicato risulti allocato, oppure zero in caso contrario.

Queste tre funzioni usano un metodo comune per scandire la mappa della memoria: il valore che rappresenta il blocco a cui si vuole fare riferimento, viene diviso per 32, ovvero il rango degli elementi dell'array che rappresenta la mappa della memoria. Il risultato intero della divisione serve per trovare quale elemento dell'array considerare, mentre il resto della divisione serve per determinare quale bit dell'elemento trovato rappresenta il blocco desiderato. Trovato ciò, si deve costruire una maschera, nella quale si mette a uno il bit che rappresenta il blocco; per farlo, si pone inizialmente a uno il bit più significativo della maschera, quindi lo si fa scorrere verso destra di

un valore pari al resto della divisione.

Per esempio, volendo individuare il terzo blocco di memoria, pari al numero 2 (il primo blocco corrisponderebbe allo zero), si avrebbe che questo è descritto dal primo elemento dell'array (in quanto $2/32$ dà zero, come risultato intero), mentre la maschera necessaria a trovare il bit corrispondente è $00100000000000000000000000000000_2$, la quale si ottiene spostando per due volte verso destra il bit più significativo (due volte, pari al resto della divisione).

Una volta determinata la maschera, per segnare come occupato un blocco di memoria, basta utilizzare l'operatore OR binario:

```
mb_table[i] = mb_table[i] | mask;
```

Se invece si vuole liberare un blocco di memoria, si utilizza un AND binario, invertendo però il contenuto della maschera:

```
mb_table[i] = mb_table[i] & ~mask;
```

Va osservato che la rappresentazione dei blocchi nella mappa è invertita rispetto ad altri sistemi operativi, in quanto non sarebbe tanto logico il fatto che il bit più significativo si riferisca invece alla parte più bassa del proprio insieme di blocchi di memoria. La scelta è dovuta al fatto che, volendo rappresentare la mappa numericamente, la lettura di questa sarebbe più vicina a quella che è la percezione umana del problema.

84.7 Dispositivi

La gestione dei dispositivi fisici, da parte di `os32`, è limitata ed essenziale. Tutte le operazioni di lettura e scrittura di dispositivi, passano attraverso la gestione comune della funzione *dev_io()*.

Nel file `lib/sys/os32.h` (listato 95.21), disponibile sia al kernel, sia alle applicazioni, sono elencate le macro-variabili che descrivono tutti i dispositivi previsti in forma numerica. Queste macro-variabili hanno nomi prefissati dalla sigla **DEV_...**. Per esempio, **DEV_DM_MAJOR** corrisponde al numero primario (*major*) per le unità di memorizzazione di massa, **DEV_DM00** corrisponde al numero primario e secondario (*major* e *minor*), in un valore unico, della prima unità di memorizzazione di massa complessiva, mentre **DEV_DM01** corrisponde alla prima partizione della stessa.

84.7.1 File «kernel/dev.h» e «kernel/dev/...»

«

Listati 94.3 e successivi.

Il file `kernel/dev.h` incorpora il file `lib/sys/os32/os32.h`, per acquisire le macro-variabili della gestione dei dispositivi che sono disponibili anche agli applicativi. Successivamente dichiara la funzione **dev_io()**, la quale sintetizza tutta la gestione dei dispositivi. Questa funzione utilizza il parametro *rw*, per specificare l'azione da svolgere (lettura o scrittura). Per questo parametro vanno usate le macro-variabili **DEV_READ** e **DEV_WRITE**, così da non dover ricordare quale valore numerico corrisponde alla lettura e quale alla scrittura.

```
ssize_t dev_io (pid_t pid, dev_t device, int rw, off_t offset,  
               void *buffer, size_t size, int *eof);
```

Sono comunque descritte anche altre funzioni, ma utilizzate esclusivamente da **dev_io()**.

La funzione **dev_io()** si limita a estrapolare il numero primario dal numero del dispositivo complessivo, quindi lo confronta con i vari

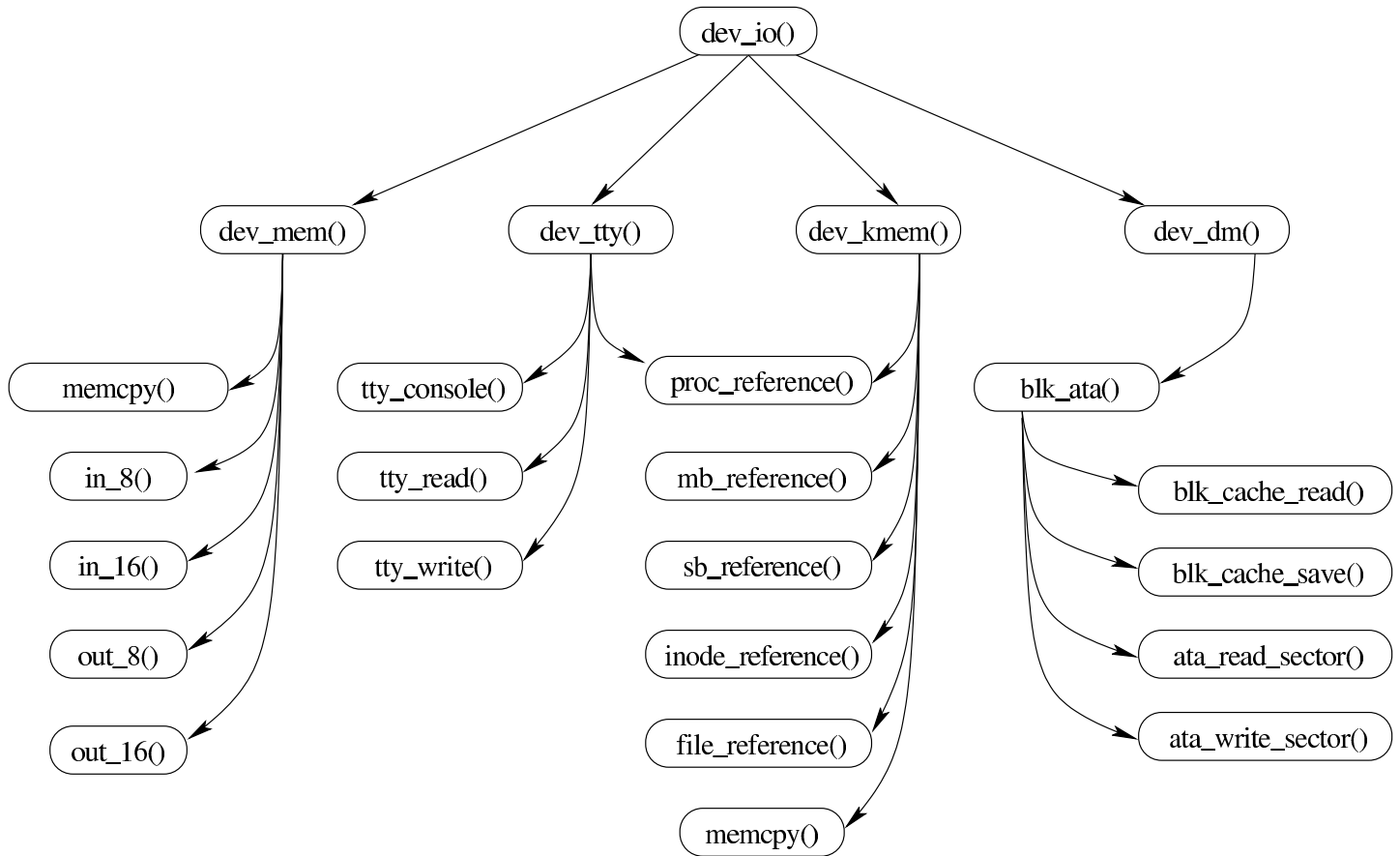
tipi gestibili. A seconda del numero primario seleziona una funzione appropriata per la gestione di quel tipo di dispositivo, passando praticamente gli stessi argomenti già ricevuti.

Va osservato il caso particolare dei dispositivi *DEV_KMEM_...*. In un sistema operativo Unix comune, attraverso ciò che fa capo al file di dispositivo `‘/dev/kmem’`, si ha la possibilità di accedere all’immagine in memoria del kernel, lasciando a un programma con privilegi adeguati la facoltà di interpretare i simboli che consentono di individuare i dati esistenti. Nel caso di os32, non ci sono simboli nel risultato della compilazione, quindi non è possibile ricostruire la collocazione dei dati. Per questa ragione, le informazioni che devono essere pubblicate, vengono controllate attraverso un dispositivo specifico. Quindi, il dispositivo *DEV_KMEM_PS* consente di leggere la tabella dei processi, *DEV_KMEM_MMAP* consente di leggere la mappa della memoria, e così vale anche per altre tabelle.

Per quanto riguarda la gestione dei terminali, attraverso la funzione *dev_tty()*, quando un processo vuole leggere dal terminale, ma non risulta disponibile un carattere, questo viene messo in pausa, in attesa di un evento legato ai terminali.

os32 gestisce virtualmente tutti i dispositivi come se fossero a caratteri. Tuttavia, nel caso delle unità di memorizzazione di massa il flusso di caratteri, in lettura o in scrittura, viene scomposto in blocchi, sfruttando anche una memoria (*cache*) per questi. Pertanto, la funzione *dev_dm()* si avvale di *blk_ata()*.

Figura 84.80. Interdipendenza tra la funzione *dev_io()* e le altre. I collegamenti con le funzioni *major()* e *minor()* sono omesse.



84.7.2 File «kernel/blk.h» e «kernel/blk/...»

«

Listati [94.2](#) e successivi.

I file contenuti nella directory ‘kernel/blk/’ riguardano specificamente la gestione della memoria *cache* per i blocchi di dati usati più di frequente, relativamente ai dispositivi di memorizzazione. In pratica, tale gestione riguarda esclusivamente le unità PATA.

La tabella *blk_table()* è composta da elementi ‘**blk_cache_t**’, ognuno dei quali rappresenta un blocco singolo, con l’indicazione del dispositivo (dell’unità intera e non di una singola partizione) e

del numero di blocco a cui si riferisce, assieme a un numero che ne rappresenta l'«età».

Inizialmente, la funzione *blk_cache_init()*, usata una volta sola all'interno di *kmain()*, si azzerano le informazioni sul numero di dispositivo e sul numero del blocco di ogni elemento della tabella, quindi si assegna l'età attraverso un numero progressivo, da 0 a *BLK_CACHE_MAX_AGE*. Il numero più basso rappresenta l'ultimo blocco letto o modificato, mentre quello più alto riguarda il blocco che da più tempo non è stato utilizzato.

Quando la funzione *blk_ata()* deve leggere un blocco da un'unità PATA, prima, attraverso la funzione *blk_cache_read()*, controlla all'interno della tabella *blk_table()* esiste già una copia del blocco; questo viene trovato, la funzione *blk_cache_read()* ne azzerà l'età, incrementando conseguentemente l'età dei blocchi che avevano prima un valore inferiore al suo. Se il blocco viene trovato nella tabella, la funzione non interpella l'hardware PATA e conclude il suo lavoro, altrimenti provvede alla lettura necessaria e al suo salvataggio nella tabella dei blocchi, con l'aiuto di *blk_cache_save()*, la quale aggiorna il blocco se questo era già presente nella tabella, oppure rimpiazza il blocco di età maggiore, aggiornando di conseguenza l'età, come nel caso della lettura.

Quando la funzione *blk_ata()* deve scrivere un blocco, la scrittura hardware avviene in ogni caso, seguita dal salvataggio nella tabella dei blocchi.

In pratica, la memoria *cache* viene usata solo per le letture, pertanto tutte le scritture sono sincrone.

84.7.3 Numero primario e numero secondario

«

I dispositivi, secondo la tradizione dei sistemi Unix, sono rappresentati dal punto di vista logico attraverso un numero intero, senza segno, a 16 bit. Tuttavia, per organizzare questa numerazione in modo ordinato, tale numero viene diviso in due parti: la prima parte, nota come *major*, ovvero «numero primario», si utilizza per individuare il tipo di dispositivo; la seconda, nota come *minor*, ovvero «numero secondario», si utilizza per individuare precisamente il dispositivo, nell'ambito del tipo a cui appartiene.

In pratica, il numero complessivo a 16 bit si divide in due, dove gli 8 bit più significativi individuano il numero primario, mentre quelli meno significativi danno il numero secondario. L'esempio seguente si riferisce al dispositivo che genera il valore zero, il quale appartiene al gruppo dei dispositivi relativi alla memoria:

DEV_MEM_MAJOR	01 ₁₆
DEV_ZERO	0104 ₁₆

In questo caso, il valore che rappresenta complessivamente il dispositivo è 0104₁₆ (pari a 260₁₀), ma si compone di numero primario 01₁₆ e di numero secondario 04₁₆ (che coincidono nella rappresentazione in base dieci). Per estrarre il numero primario si deve dividere il numero complessivo per 256 (0100₁₆), trattenendo soltanto il risultato intero; per filtrare il numero secondario si può fare la stessa divisione, ma trattenendo soltanto il resto della stessa. Al contrario, per produrre il numero del dispositivo, partendo dai numeri primario e secondario separati, occorre moltiplicare il numero primario per

256, sommando poi il risultato al numero secondario.

84.7.4 Dispositivi previsti

L'astrazione della gestione dei dispositivi, consente di trattare tutti i componenti che hanno a che fare con ingresso e uscita di dati, in modo sostanzialmente omogeneo; tuttavia, le caratteristiche effettive di tali componenti può comportare delle limitazioni o delle peculiarità. Ci sono alcune questioni fondamentali da considerare: un tipo di dispositivo potrebbe consentire l'accesso in un solo verso (lettura o scrittura); l'accesso al dispositivo potrebbe essere ammesso solo in modo sequenziale, rendendo inutile l'indicazione di un indirizzo; la dimensione dell'informazione da trasferire potrebbe assumere un significato differente rispetto a quello comune.

Tabella 84.82. Classificazione dei dispositivi di os32.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_MEM	r/w	diret- to	Permette l'accesso alla memo- ria, in modo indiscriminato; tut- tavia, solo al kernel è permessa la scrittura.
DEV_NULL	r/w	nes- suno	Consente la lettura e la scrittura, ma non si legge e non si scrive alcunché.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_PORT	r/w	se- quen- ziale	Consente di leggere e scrivere da o verso una porta di I/O, individuata attraverso l'indirizzo di accesso (l'indirizzo, o meglio lo scostamento, viene trattato come la porta a cui si vuole accedere). Tuttavia, la dimensione dell'informazione da trasferire è valida solo se si tratta di uno o di due byte: per la dimensione di un byte si usano le funzioni <i>in_8()</i> e <i>out_8()</i> ; per due byte si usano le funzioni <i>in_16()</i> e <i>out_16()</i> . Per dimensioni differenti la lettura o la scrittura non ha effetto.
DEV_ZERO	r	se- quen- ziale	Consente solo la lettura di valori a zero (zero inteso in senso binario).
DEV_TTY	r/w	se- quen- ziale	Rappresenta il terminale virtuale del processo attivo.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_DM <i>m n</i>	r/w	diret- to	Rappresenta la partizione <i>n</i> dell'unità di memorizzazione <i>m</i> . La prima unità PATA disponibile ottiene il dispositivo <i>DEV_DM00</i> , la seconda il numero <i>DEV_DM10</i> , ecc.
DEV_KMEM_PS	r	diret- to	Rappresenta la tabella contenente le informazioni sui processi. L'indirizzo di accesso indica il numero del processo di partenza; la dimensione da leggere dovrebbe essere abbastanza grande da contenere un processo, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
DEV_KMEM_MMP	r	se- quen- ziale	Rappresenta la mappa della memoria, alla quale si può accedere solo dal suo principio. In pratica, l'indirizzo di accesso viene ignorato, mentre conta solo la quantità di byte richiesta.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_KMEM_SB	r	diret- to	Rappresenta la tabella dei su- per blocchi (per la gestione delle unità di memorizzazio- ne). L'indirizzo di accesso ser- ve a individuare il super blocco; la dimensione richiesta dovreb- be essere abbastanza grande da contenere un super blocco, ma anche richiedendo una dimen- sione maggiore, se ne legge uno solo.
DEV_KMEM_INODE	r	diret- to	Rappresenta la tabella degli ino- de (per la gestione delle unità di memorizzazione). L'indiriz- zo di accesso serve a individua- re l'inode; la dimensione richie- sta dovrebbe essere abbastanza grande da contenere un inode, ma anche richiedendo una di- mensione maggiore, se ne legge uno solo.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_KMEM_FILE	r	diret- to	Rappresenta la tabella dei file (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare il file; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di un file, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
DEV_KMEM_ARP	r	diret- to	Rappresenta la tabella ARP (per la trasformazione degli indirizzi IPv4 in indirizzi Ethernet). L'indirizzo di accesso serve a individuare la voce; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di una voce, ma anche richiedendo una dimensione maggiore, se ne legge una sola.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_KMEM_NET	r	diret- to	Rappresenta la tabella delle interfacce di rete. L'indirizzo di accesso serve a individuare la voce della tabella; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di una voce, ma anche richiedendo una dimensione maggiore, se ne legge una sola.
DEV_KMEM_ROUTE	r	diret- to	Rappresenta la tabella degli instradamenti IPv4. L'indirizzo di accesso serve a individuare la voce della tabella; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di una voce, ma anche richiedendo una dimensione maggiore, se ne legge una sola.
DEV_CONSOLE	r/w	se- quen- ziale	Legge o scrive relativamente alla console attiva la quantità di byte richiesta, ignorando l'indirizzo di accesso.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_CONSOLE n	r/w	se- quen- ziale	Legge o scrive relativamente alla console n la quantità di byte richiesta, ignorando l'indirizzo di accesso.

84.7.5 Gestione del terminale

Listato [94.4.42](#) e successivi.

Il terminale offre solo la funzionalità elementare della modalità canonica, dove è possibile scrivere o leggere sequenzialmente. Ci sono al massimo quattro terminali virtuali, selezionabili attraverso le combinazioni di tasti [*Ctrl q*], [*Ctrl r*], [*Ctrl s*] e [*Ctrl t*] e non è possibile controllare i colori o la posizione del testo che si va a esporre; in pratica si opera come su una telescrivente. Le funzioni di livello più basso, relative al terminale hanno nomi che iniziano per 'tty_... ()'.

Per la gestione dei quattro terminali virtuali, si utilizza una tabella, in cui ogni voce rappresenta lo stato del terminale virtuale che rappresenta. La tabella è costituita dall'array *tty_table[]* che contiene *TTY_TOTALS* elementi. L'array è dichiarato nel file 'kernel/driver/tty_public.c', mentre la macro-variabile *TTY_TOTALS* appare nel file 'kernel/driver/tty.h'. Gli elementi di *tty_table[]* sono di tipo 'tty_t':

```

typedef struct {
    dev_t          device;
    pid_t          pgrp;           // Process group.
    struct termios attr;         // termios attributes.
    unsigned char  status;       // 0 = edit,
                                // 1 = end edit.
    char           line[MAX_CANON]; // Canonical input line.
    int            lpr;           // Input line position
                                // read.
    int            lpw;           // Input line position
                                // write.
} tty_t;

```

Il membro *attr* della voce di un terminale è una variabile strutturata di tipo ‘**struct termios**’, come previsto nel file ‘`termios.h`’ della libreria standard.

L’input del terminale, proveniente dalla tastiera, viene depositato dalla funzione *proc_sch_terminals()* all’interno del membro *line[]*, annotando in *lpw* l’indice di scrittura. Quando si legge dal terminale, si ottiene un carattere alla volta da *line[]*, con l’ausilio dell’indice *lpr*. Quando il terminale virtuale riceve input dalla tastiera, è nello stato definito dalla macro-variabile *TTY_INPUT_LINE_EDITING*, mentre quando l’inserimento risulta concluso, per esempio perché è stato premuto il tasto [*Invio*], lo stato è quello di *TTY_INPUT_LINE_CLOSED* ed è possibile procedere con la lettura del contenuto di *line[]*: quando la lettura termina perché l’indice *lpr* ha raggiunto *lpw*, gli indici vengono azzerati e lo stato ritorna quello di inserimento. Quando un processo tenta di leggere dal terminale, mentre questo è in fase di inserimento, non ancora concluso, viene sospeso e rimane così fino alla conclusione dell’inserimento stesso.

Figura 84.84. A sinistra le fasi dell'inserimento di una riga da tastiera; a destra le fasi della lettura attraverso la funzione `tty_read()`.

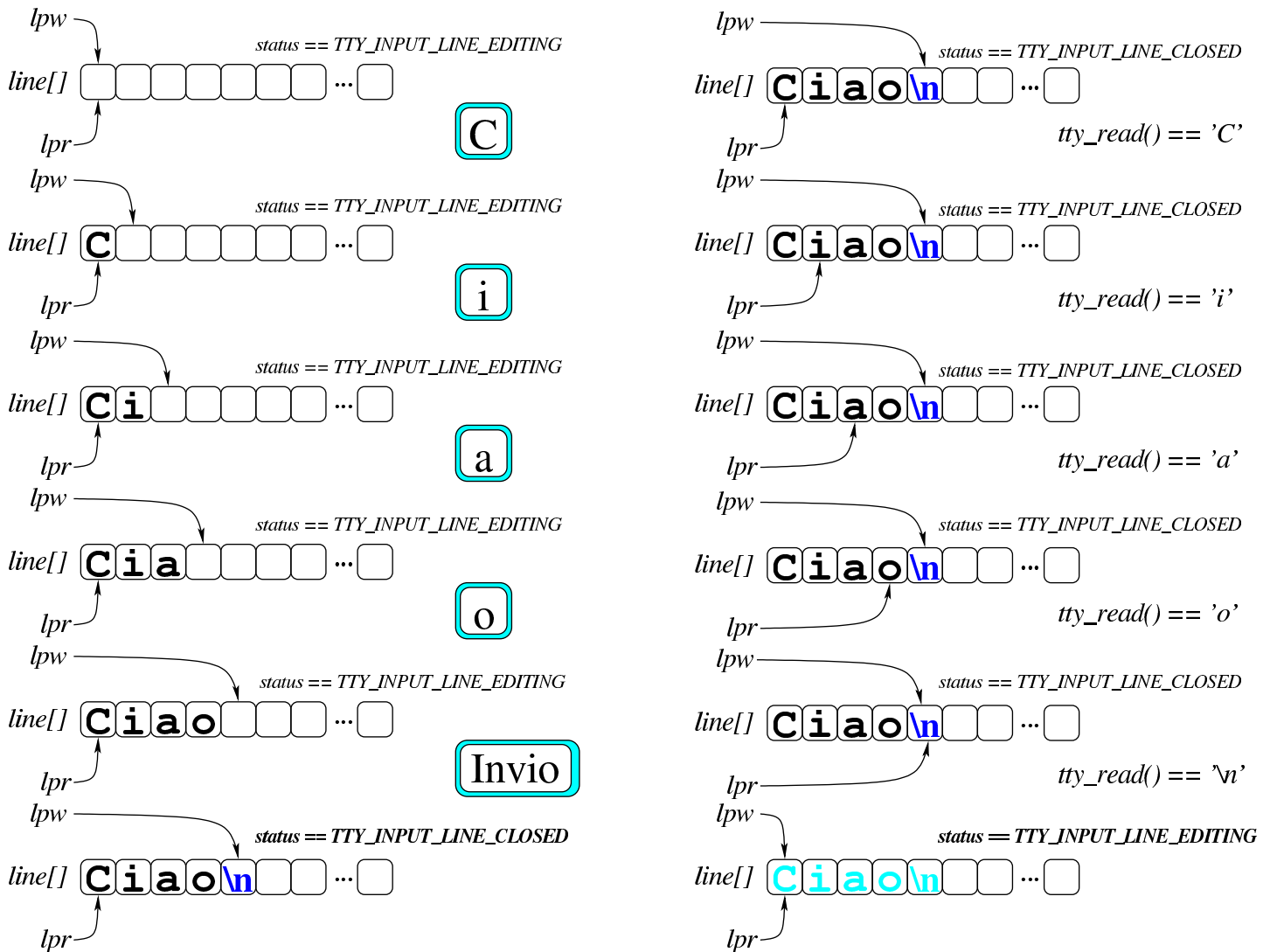


Tabella 84.85. Funzioni per l'accesso al terminale, dichiarate nel file di intestazione `'kernel/driver/tty.h'` e descritte nei file contenuti nella directory `'kernel/driver/tty/'`.

Funzione	Descrizione
<pre>dev_t tty_console (dev_t <i>device</i>);</pre>	Seleziona un terminale virtuale, rendendolo attivo, specificandone il numero del dispositivo. La funzione restituisce il dispositivo attivo in precedenza e se le viene fornito solo il valore zero, il terminale virtuale non cambia, ma si ottiene comunque di conoscere qual è quello attuale.
<pre>void tty_init (void);</pre>	Inizializza la gestione dei terminali virtuali, popolando anche la tabella <i>tty_table[]</i> con i valori predefiniti. Questa funzione viene usata una volta sola all'interno di <i>kmain()</i> .
<pre>int tty_read (dev_t <i>device</i>);</pre>	Legge un carattere dal terminale virtuale specificato attraverso il numero di dispositivo. La lettura avviene solo se l'input da tastiera risulta concluso, altrimenti la funzione restituisce il valore -1.

Funzione	Descrizione
<pre>tty_t *tty_reference (dev_t <i>device</i>);</pre>	<p>Restituisce il puntatore alla voce della tabella <i>tty_table[]</i> contenente le informazioni sul terminale virtuale indicato attraverso il numero di dispositivo. Se il dispositivo indicato non è valido, si ottiene il puntatore nullo; se viene richiesto il dispositivo indefinito, si ottiene il puntatore all'inizio della tabella.</p>
<pre>void tty_write (dev_t <i>device</i>, int <i>c</i>);</pre>	<p>Scrive un carattere sullo schermo del terminale specificato.</p>

84.7.5.1 Gestione della tastiera

Listato [94.4.15](#) e successivi.

Per la gestione della tastiera, nel file 'kernel/driver/kbd_public.c' viene dichiarata una variabile strutturata, di tipo '**kbd_t**', contenente le informazioni sullo stato della stessa e sulla mappa di trasformazione da applicare. A differenza della gestione complessiva dei terminali, in cui ogni terminale virtuale ha un proprio insieme di dati, per la tastiera questo è unico.



Listato 84.86. Definizione del tipo `'kbd_t'`, contenuto nel file `'kernel/driver/kbd.h'`.

```
typedef struct {
    bool        shift;
    bool        shift_lock;
    bool        ctrl;
    bool        alt;
    bool        echo;
    unsigned char key;
    unsigned char map1[128];
    unsigned char map2[128];
} kbd_t;
```

Nella variabile *kbd*, come si intuisce dai nomi dei suoi membri, viene annotato lo stato di pressione dei tasti delle maiuscole, dei tasti *[Ctrl]* e *[Alt]*, per poter recepire eventuali combinazioni di tasti; inoltre, il membro *echo*, se attivo, indica la richiesta di vedere sullo schermo ciò che si digita.

Dalla tastiera viene recepito un solo tasto alla volta: se questo si traduce in un carattere, stampabile o meno che sia, questo viene depositato nel membro *key*, da dove la funzione *proc_sch_terminals()* deve provvedere a prelevarlo (per trasferirlo nel membro *line[]* della voce che descrive il terminale virtuale attivo), azzerando nuovamente *key*. Fino a quando il membro *key* ha un valore diverso da zero, non è possibile recepire altro dalla tastiera.

Dalla tastiera è possibile ottenere solo i caratteri ASCII; in particolare, quelli non stampabili si ottengono per combinazione con il tasto *[Ctrl]*, secondo la convenzione tradizionale. Non sono previste altre funzionalità.

Tabella 84.87. Funzioni per la gestione della tastiera, dichiarate nel file di intestazione ‘kernel/driver/kbd.h’ e descritte nei file contenuti nella directory ‘kernel/driver/kbd/’.

Funzione	Descrizione
<pre>void kbd_isr (void);</pre>	<p>Questa funzione è chiamata dalla routine di gestione delle interruzioni da tastiera, contenuta nel file ‘kernel/ibm_i386/isr.s’. La funzione legge un carattere dalla porta di I/O 60₁₆, quindi lo interpreta e aggiorna il contenuto della variabile strutturata <i>kbd</i> di conseguenza.</p>
<pre>void kbd_load (void);</pre>	<p>Questa funzione è chiamata una sola volta da <i>kmain()</i>, per associare la mappa della tastiera ai codici prodotti dalla stessa. Attualmente questa funzione produce esclusivamente l’associazione necessaria per una tastiera italiana.</p>

La funzione *proc_scheduler()*, il cui scopo principale è quello di alternare i processi in esecuzione, tra le altre cose, avvia ogni volta la funzione *proc_scheduler_terminals()*. La funzione *proc_scheduler_terminals()* verifica se nella variabile *kbd.key* è disponibile un valore diverso da zero e, se c’è, lo acquisisce per conto del terminale attivo. Prima di tutto verifica se si tratta di una combinazione di tasti che richiede lo scambio a un altro terminale virtuale; poi controlla se si tratta di un codice di interruzione (come quello provocato da [Ctrl c]) e, se la configurazione del terminale

attivo lo permette, conclude il processo più interno appartenente al gruppo che risulta connesso al terminale stesso; alla fine, dopo altre ipotesi particolari, se si tratta di un carattere «normale» e il terminale si trova in fase di inserimento (*TTY_INPUT_LINE_EDITING*), questo viene depositato nell'array *line[]*, con il conseguente aggiornamento dell'indice di scrittura al suo interno; ricevendo invece un codice che rappresenta la conclusione dell'inserimento, si rimette il terminale nello stato di conclusione dell'inserimento (*TTY_INPUT_LINE_CLOSED*).

84.7.5.2 Gestione dello schermo

«

Listato [94.4.30](#) e successivi.

Lo schermo di os32 viene gestito secondo quanto prescrive l'hardware VGA (come descritto nella sezione [83.3](#)), per cui ciò che si vuole fare apparire deve essere scritto in memoria a partire dall'indirizzo $B8000_{16}$, usando per ogni carattere 16 bit (8 bit di questo gruppo servono per gli attributi).

Dal momento che si gestiscono dei terminali virtuali, per ognuno di questi occorre tenere una copia dell'immagine dello schermo, così, quando si seleziona un terminale differente, la copia di quel terminale viene usata per sovrascrivere l'area di memoria che rappresenta lo schermo. Per la gestione degli schermi virtuali si usa una tabella, denominata *screen_table[]*, composta da voci di tipo 'screen_t'.

Listato 84.88. Definizione del tipo `'screen_t'`, contenuto nel file `'kernel/driver/screen.h'`.

```
typedef struct {
    uint16_t  cell[SCREEN_CELLS];
    int       position;
} screen_t;
```

All'interno della struttura rappresentata dal tipo `'screen_t'`, si vede un array che riproduce la rappresentazione in memoria dello stesso, da copiare a partire dall'indirizzo $B8000_{16}$, quando lo schermo virtuale diventa quello attivo; inoltre si vede il membro *position*, usato per ricordare la posizione in cui si trova il cursore.

Tabella 84.89. Funzioni per la gestione dello schermo, dichiarate nel file di intestazione `'kernel/driver/screen.h'` e descritte nei file contenuti nella directory `'kernel/driver/screen/'`.

Funzione	Descrizione
<code>int screen_clear (screen_t *<i>screen</i>);</code>	Ripulisce il contenuto dello schermo selezionato, riposizionando il cursore all'inizio.
<code>screen_t *screen_current (void);</code>	Restituisce il puntatore alla voce della tabella <i>screen_table[]</i> che descrive lo schermo virtuale attivo.
<code>void screen_init (void);</code>	Inizializza la gestione degli schermi virtuali, ripulendoli e collocando il cursore all'inizio. Questa funzione viene usata da <i>tty_init()</i> .

Funzione	Descrizione
<pre>int screen_newline (screen_t *<i>screen</i>);</pre>	<p>Produce sullo schermo virtuale selezionato un avanzamento alla riga successiva. Ciò può comportare semplicemente il riposizionamento del cursore, oppure lo scorrimento in avanti del contenuto, quando il cursore si trova già sull'ultima riga visualizzabile.</p>
<pre>int screen_number (screen_t *<i>screen</i>);</pre>	<p>Restituisce il numero dello schermo corrispondente al puntatore fornito, purché questo sia valido. È in pratica l'opposto della funzione <i>screen_pointer()</i>.</p>
<pre>screen_t *screen_pointer (int <i>scrn</i>);</pre>	<p>Restituisce il puntatore alla voce della tabella <i>screen_table[]</i> che descrive lo schermo virtuale indicato per numero. È in pratica l'opposto della funzione <i>screen_number()</i>.</p>

Funzione	Descrizione
<pre>int screen_putc (screen_t *screen, int c);</pre>	<p>Colloca sullo schermo virtuale individuato dal puntatore che costituisce il primo parametro, il carattere richiesto come secondo. Se il carattere in questione è <CR> o <LF>, si produce un avanzamento alla riga successiva, mentre con un carattere <BS> si produce un arretramento del cursore.</p>
<pre>uint16_t screen_cell (c, <i>attributo</i>);</pre>	<p>Si tratta di una macroistruzione che produce il valore corretto per una cella dello schermo VGA, contenente sia l'informazione sul carattere, sia quella dell'attributo associato.</p>
<pre>int screen_scroll (screen_t *screen);</pre>	<p>Fa scorrere in avanti lo schermo, di una riga, ricollocando di conseguenza il cursore.</p>
<pre>int screen_select (screen_t *screen);</pre>	<p>Seleziona lo schermo indicato come schermo attivo, facendone apparire il contenuto sullo schermo VGA reale.</p>

Funzione	Descrizione
<pre>void screen_update (screen_t *screen);</pre>	<p>Aggiorna la memoria VGA sulla base della copia che rappresenta lo schermo virtuale attivo. L'aggiornamento implica anche la collocazione del cursore visibile in corrispondenza della posizione attuale.</p>

84.7.5.3 Configurazione del terminale

«

Lo standard dei sistemi Unix prescrive che per ogni terminale gestito sia prevista una variabile strutturata, di tipo *struct termios*, allo scopo di contenere la configurazione dello stesso. os32 gestisce i terminali virtuali soltanto in modalità «canonica», ovvero come se si trattasse di telescriventi, anche se munite di video invece che di carta, pertanto utilizza solo un sottoinsieme delle opzioni previste.

```
typedef uint16_t      tcflag_t;
typedef unsigned char cc_t;
...
struct termios {
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    cc_t      c_cc[NCCS];
};
```

Il membro *c_cc[]* è un array di caratteri di controllo, a cui viene attribuita una definizione.

Tabella 84.91. Caratteri di controllo riconosciuti da `os32`, secondo le definizioni del file `'termios.h'`.

Definizione	Corrispondenza	Descrizione
<i>VEOF</i>	04 ₁₆ <EOT>	Carattere di fine file.
<i>VERASE</i>	08 ₁₆ <BS>	Carattere di cancellazione.
<i>VINTR</i>	03 ₁₆ <ETX>	Carattere di interruzione.
<i>VQUIT</i>	1C ₁₆ <FS>	Carattere di abbandono.

Il membro ***c_iflag*** serve a contenere opzioni sull'inserimento, ovvero sul controllo della digitazione.

Tabella 84.92. Opzioni del membro ***c_iflag*** riconosciute da `os32`.

Opzione	Descrizione
<i>BRKINT</i>	Se questa opzione è attiva e, nel contempo, non è attiva <i>IGNBRK</i> , si intendono recepire i codici di interruzione <i>VINTR</i> . Se l'opzione <i>ISIG</i> del membro <i>c_lflag</i> è attiva, il processo più interno del gruppo a cui appartiene il terminale viene concluso; in ogni caso, viene annullato il contenuto della riga di inserimento in corso.
<i>ICRNL</i>	Se si riceve il carattere <CR>, questo viene convertito in <NL>.
<i>IGNBRK</i>	Se questa opzione è attiva, fa sì che il carattere definito come <i>VINTR</i> sia ignorato.
<i>IGNCR</i>	Se si riceve il carattere <CR>, questo viene ignorato semplicemente.
<i>INLCR</i>	Se si riceve il carattere <NL>, questo viene convertito in <CR>.

Il membro ***c_lflag*** serve a contenere delle opzioni definite come «locali», le quali si occupano in pratica di controllare la visualizzazione della digitazione introdotta e di decidere se l'interruzione ricevuta da tastiera debba produrre l'invio di un segnale di interruzione al processo con cui si sta interagendo. Gli altri due membri della struttura

non vengono utilizzati da os32.

Tabella 84.93. Opzioni del membro *c_lflag* riconosciute da os32.

Opzione	Descrizione
<i>ECHO</i>	Abilita la visualizzazione sullo schermo del testo inserito da tastiera.
<i>ECHOE</i>	Ammesso che sia attiva l'opzione <i>ECHO</i> , questa abilita il recepimento del carattere definito come <i>VERASE</i> per cancellare l'ultimo carattere inserito, indietro di una posizione.
<i>ECHONL</i>	Indipendentemente dall'opzione <i>ECHO</i> , questa abilita il recepimento del carattere <code><NL></code> per fare avanzare il cursore alla riga successiva, con l'eventuale scorrimento in avanti se si trova già sull'ultima.
<i>ISIG</i>	Ammesso che sia recepito e accettato un codice di interruzione, definito come <i>VINTR</i> , con questa opzione si ottiene l'invio di un segnale di interruzione al processo più interno del gruppo collegato al terminale (il processo più interno dovrebbe corrispondere a quello in primo piano al momento della digitazione).

84.7.6 Gestione delle unità di memorizzazione in generale

« Listato [94.4](#) e successivi.

Le unità di memorizzazione vengono viste da os32 attraverso un gruppo di dispositivi astratti, definiti come ***DEV_DM****, dove la prima unità PATA disponibile ottiene il numero ***DEV_DM00***, la seconda ***DEV_DM10***,...

Il numero del dispositivo che rappresenta queste unità è composto in modo solito per ciò che riguarda la distinzione tra numero primario e numero secondario, ma il numero secondario si scompone

ulteriormente in due parti: l'unità intera e la partizione. Per esempio, il numero 0810_{16} individua la seconda unità di memorizzazione per intero, mentre 0811_{16} rappresenta la prima partizione della seconda unità.

Le unità di memorizzazione riconosciute dal sistema sono raccolte in una tabella, denominata *dm_table[]*. Ogni elemento di questa tabella contiene l'informazione sul tipo di unità, un puntatore per raggiungere un'altra tabella con le informazioni specifiche sull'unità, in base al tipo di questa (attualmente l'unica tabella in questione può essere quella delle unità PATA), le informazioni sulle partizioni esistenti (ma solo quelle primarie).

Inizialmente, all'interno di *proc_init()*, viene avviata la funzione *dm_init()*, la quale a sua volta scandisce le unità PATA attraverso *ata_init()* e ne raccoglie le informazioni nella propria tabella *dm_table()*.

84.7.7 Gestione delle unità PATA

Listato [94.4.3](#) e successivi.

La gestione delle unità PATA di os32 si limita alla modalità PIO (*programmed input-output*), con accesso LBA28, senza nemmeno considerare le partizioni. La spiegazione sul come avvenga la gestione di un'unità PATA, secondo le stesse modalità usate da os32 è disponibile nella sezione [83.9](#).

Per la gestione delle unità PATA, os32 utilizza una tabella, denominata *ata_table[]*, composta da voci di tipo *ata_t*, ognuna delle quali contiene lo stato di un'unità. L'indice della tabella corrisponde al numero dell'unità, ovvero al parametro *drive* di varie funzioni. La tabella è dichiarata formalmente nel file `kernel/driver/ata/`

ata_public.c’, mentre il tipo derivato ‘ata_t’ è descritto nel file ‘kernel/driver/ata.h’. Per comodità, si trova anche il tipo ‘ata_sector_t’, usato per descrivere lo spazio di memoria usato per collocare la copia di un settore di dati di un’unità PATA.

Tabella 84.94. Funzioni per la gestione delle unità PATA, dichiarate nel file di intestazione ‘kernel/driver/ata.h’ e descritte nei file contenuti nella directory ‘kernel/driver/ata/'. Le funzioni sono raggruppate in insiemi logici.

Funzione	Descrizione
<pre>void ata_init (void);</pre>	Inizializza la gestione delle unità PATA, predisponendo i contenuti della tabella <i>ata_table[]</i> , verificando la presenza delle unità. Questa funzione viene usata una volta sola, nella funzione <i>proc_init()</i> .
<pre>void ata_reset (int <i>drive</i>);</pre>	Azzera lo stato di funzionamento dell’unità PATA specificata.
<pre>int ata_valid (int <i>drive</i>);</pre>	Verifica se l’unità richiesta è presente effettivamente. In caso di successo restituisce il valore zero, altrimenti si ottiene -1.

Funzione	Descrizione
<pre> int ata_cmd_identify_device (int <i>drive</i>, void *<i>buffer</i>); </pre>	<p>Richiede all'unità specificata le informazioni sulla sua identificazione. Se l'unità è presente, in corrispondenza del puntatore fornito si ottengono le informazioni nello spazio di un settore (ATA_SECTOR_SIZE); l'analisi successiva di questi dati può dare maggiori informazioni sull'unità.</p>
<pre> int ata_cmd_read_sectors (int <i>drive</i>, unsigned int <i>sector</i>, unsigned char <i>count</i>, void *<i>buffer</i>); </pre>	<p>Legge dall'unità <i>drive</i>, a partire dal settore <i>sector</i>, una quantità pari a <i>count</i> settori, mettendo il risultato a partire dall'indirizzo di memoria <i>buffer</i>. Se <i>count</i> fosse pari a zero, si intenderebbero 256 settori. Se l'operazione fallisce, restituisce un valore negativo.</p>

Funzione	Descrizione
<pre>int ata_cmd_write_sectors (int <i>drive</i>, unsigned int <i>sector</i>, unsigned char <i>count</i>, void *<i>buffer</i>);</pre>	<p>Scrive nell'unità <i>drive</i>, a partire dal settore <i>sector</i>, una quantità pari a <i>count</i> settori, leggendoli a partire dall'indirizzo di memoria <i>buffer</i>. Se <i>count</i> fosse pari a zero, si intenderebbero 256 settori. Se l'operazione fallisce, restituisce un valore negativo.</p>
<pre>int ata_device (int <i>drive</i>, unsigned int <i>sector</i>);</pre>	<p>Imposta il registro <i>device</i> dell'unità PATA specificata, con l'indicazione di un numero di settore.</p>

Funzione	Descrizione
<pre>int ata_rdy (int <i>drive</i>, clock_t <i>timeout</i>);</pre>	<p>Attende che l'unità <i>drive</i> sia pronta, purché ciò avvenga entro il tempo <i>timeout</i>. Se l'operazione ha successo, la funzione restituisce zero, altrimenti dà un valore negativo.</p>
<pre>int ata_drq (int <i>drive</i>, clock_t <i>timeout</i>);</pre>	<p>Attende che l'unità <i>drive</i> sia pronta a ricevere dati, purché ciò avvenga entro il tempo <i>timeout</i>. Se l'operazione ha successo, la funzione restituisce zero, altrimenti dà un valore negativo.</p>

Funzione	Descrizione
<pre>int ata_lba28 (int <i>drive</i>, unsigned int <i>sector</i>, unsigned char <i>count</i>);</pre>	<p>Invia all'unità <i>drive</i> la prima parte di un comando, in cui sono contenute le coordinate LBA28.</p>

Funzione	Descrizione
<pre>int ata_read_sector (int <i>drive</i>, unsigned int <i>sector</i>, void *<i>buffer</i>);</pre>	<p>È una macroistruzione che legge dall'unità <i>drive</i>, il settore <i>sector</i>, mettendo il risultato a partire dall'indirizzo di memoria <i>buffer</i>. La macroistruzione si avvale praticamente della funzione <i>ata_cmd_read_sectors()</i>, per leggere un solo settore.</p>
<pre>int ata_write_sector (int <i>drive</i>, unsigned int <i>sector</i>, void *<i>buffer</i>);</pre>	<p>È una macroistruzione che scrive nell'unità <i>drive</i>, il settore <i>sector</i>, traendo i dati dall'indirizzo di memoria <i>buffer</i>. La macroistruzione si avvale praticamente della funzione <i>ata_cmd_write_sectors()</i>, per scrivere un solo settore.</p>

84.8 Gestione del file system

«

La gestione del file system è suddivisa in diversi file contenuti nella directory `'kernel/fs/'`, facenti capo al file di intestazione `'kernel/fs.h'`.

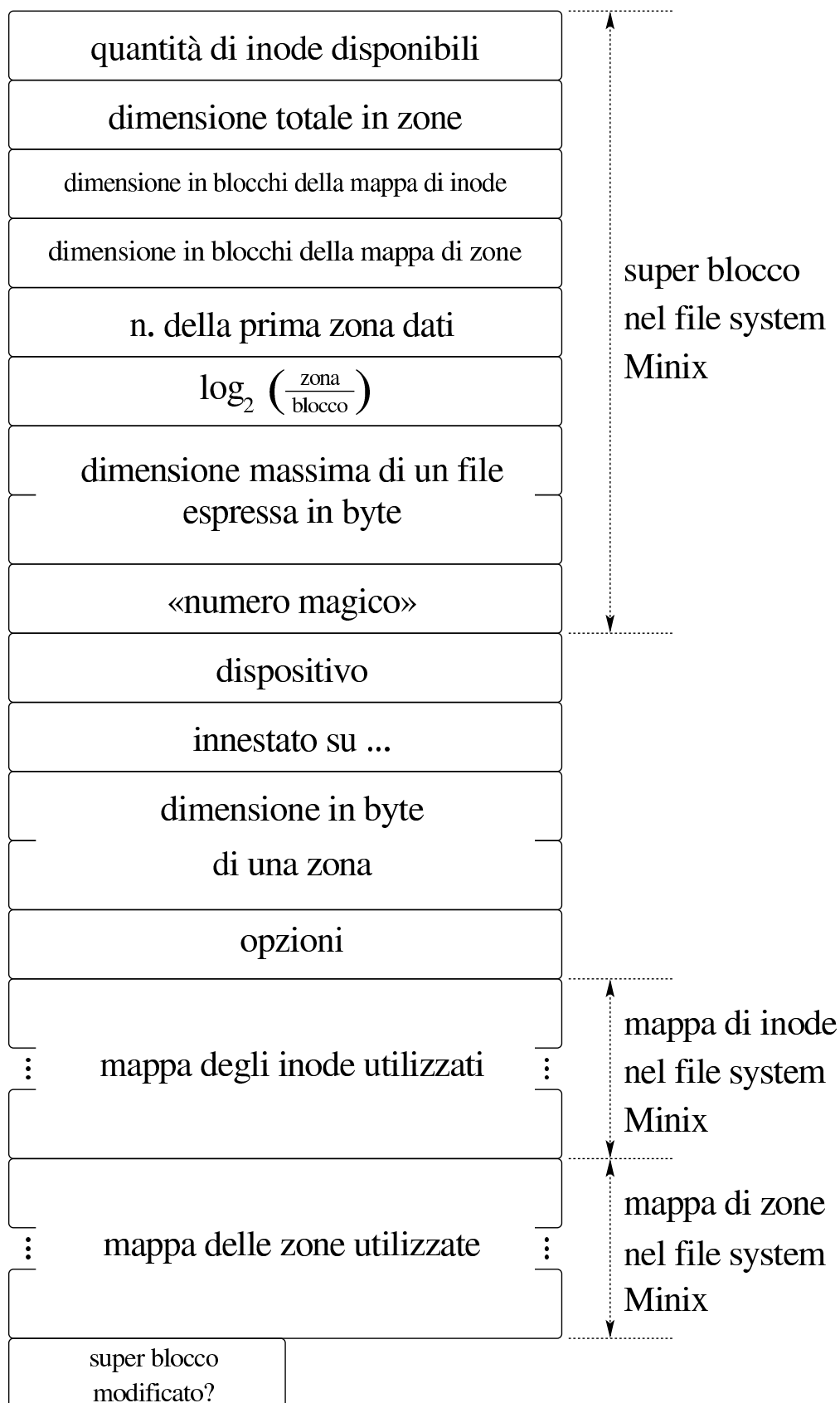
Listato [94.5](#) e successivi.

84.8.1 File «kernel/fs/sb_...»

«

I file `'kernel/fs/sb_...'` descrivono le funzioni per la gestione dei super blocchi, distinguibili perché iniziano tutte con il prefisso `'sb_'`. Tra questi file si dichiara l'array `sb_table[]`, il quale rappresenta una tabella le cui righe sono rappresentate da elementi di tipo `'sb_t'` (il tipo `'sb_t'` è definito nel file `'kernel/fs.h'`). Per uniformare l'accesso alla tabella, la funzione `sb_reference()` permette di ottenere il puntatore a un elemento dell'array `sb_table[]`, specificando il numero del dispositivo cercato.

Figura 84.95. Struttura del tipo 'sb_t', corrispondente agli elementi dell'array *sb_table[]*.



Listato 84.96. Struttura del tipo `'sb_t'`, corrispondente agli elementi dell'array `sb_table[]`.

```
typedef struct sb          sb_t;

struct sb {
    uint16_t  inodes;
    uint16_t  zones;
    uint16_t  map_inode_blocks;
    uint16_t  map_zone_blocks;
    uint16_t  first_data_zone;
    uint16_t  log2_size_zone;
    uint32_t  max_file_size;
    uint16_t  magic_number;
    //-----
    dev_t     device;
    inode_t   *inode_mounted_on;
    blksize_t blksize;
    int       options;
    uint16_t  map_inode[SB_MAP_INODE_SIZE];
    uint16_t  map_zone[SB_MAP_ZONE_SIZE];
    char      changed;
};
```

Il super blocco rappresentato dal tipo `'sb_t'` include anche le mappe delle zone e degli inode impegnati. Queste mappe hanno una dimensione fissa in memoria, mentre nel file system reale possono essere di dimensione minore. La tabella di super blocchi, contiene le informazioni dei dispositivi di memorizzazione innestati nel sistema. L'innesto si concretizza nel riferimento a un inode, contenuto nella tabella degli inode (descritta in un altro capitolo), il quale rappresenta la directory di un'altra unità, su cui tale innesto è avvenuto. Naturalmente, l'innesto del file system principale rappresenta un caso particolare.

Tabella 84.97. Funzioni per la gestione dei dispositivi di memorizzazione di massa, a livello di super blocco, definite nei file ‘kernel/fs/sb_...’.

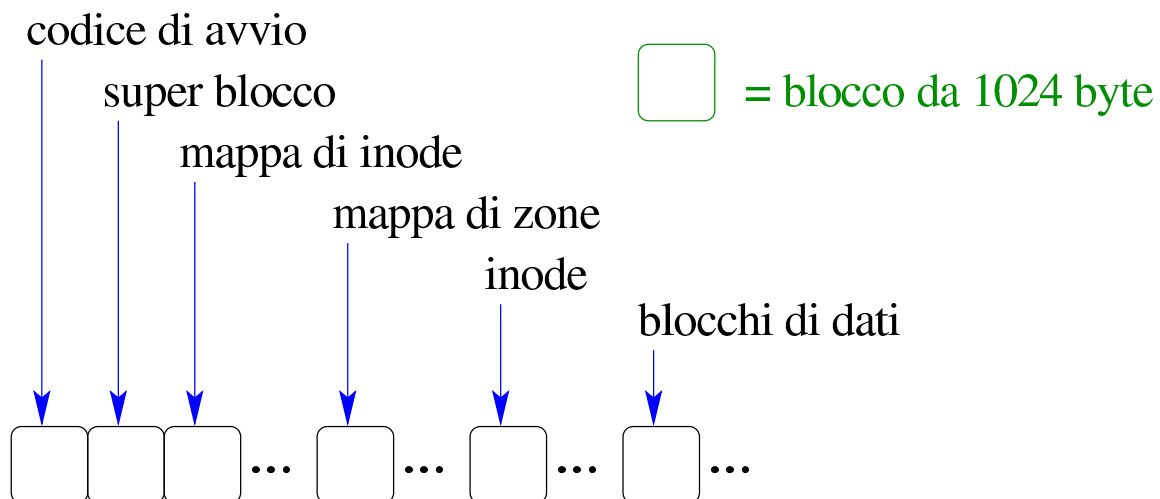
Funzione	Descrizione
<pre>sb_t *sb_reference (dev_t <i>device</i>);</pre>	<p>Restituisce il riferimento a un elemento della tabella dei super blocchi, in base al numero del dispositivo di memorizzazione. Se il dispositivo cercato non risulta già innestato, si ottiene il puntatore nullo; se si chiede il dispositivo zero, si ottiene il puntatore al primo elemento della tabella.</p>

Funzione	Descrizione
<pre>sb_t *sb_mount (dev_t <i>device</i>, inode_t **<i>inode_mnt</i>, int <i>options</i>);</pre>	<p>Innesta il dispositivo rappresentato numericamente dal primo parametro, sulla directory corrispondente all'inode a cui punta il secondo parametro, con le opzioni del terzo parametro. Quando si tratta del primo innesto del file system principale, la directory è quella dello stesso file system, pertanto, in tal caso, <i>*inode_mnt</i> è inizialmente un puntatore nullo e deve essere modificato dalla funzione stessa.</p>
<pre>int sb_save (sb_t *<i>sb</i>);</pre>	<p>Salva il super blocco nella sua unità di memorizzazione, se questo risulta modificato. In questo caso, il super blocco include anche le mappe degli inode e delle zone.</p>
<pre>int sb_zone_status (sb_t *<i>sb</i>, zno_t <i>zone</i>);</pre>	<p>Restituisce uno se la zona rappresentata dal secondo parametro è impegnata nel super blocco a cui si riferisce il primo parametro; diversamente restituisce zero.</p>

Funzione	Descrizione
<pre>int sb_inode_status (sb_t *sb, ino_t ino);</pre>	Restituisce uno se l'inode rappresentato dal secondo parametro è impegnato nel super blocco a cui si riferisce il primo parametro; diversamente restituisce zero.
<pre>void sb_print (void);</pre>	Funzione diagnostica per la visualizzazione sullo schermo dello stato della tabella dei super blocchi.

84.8.2 File «kernel/fs/zone_...»

Nel file system Minix 1, si distinguono i concetti di blocco e zona di dati, con il vincolo che la zona ha una dimensione multipla del blocco. Il contenuto del file system, dopo tutte le informazioni amministrative, è organizzato in zone; in altri termini, i blocchi di dati si raggiungono in qualità di zone. «



La zona rimane comunque un tipo di blocco, potenzialmente più grande (ma sempre multiplo) del blocco vero e proprio, che si numerava a partire dall'inizio dello spazio disponibile, con la differenza che è utile solo per raggiungere i blocchi di dati. Nel super blocco del file system si trova l'informazione del numero della prima zona che contiene dati, in modo da non dover ricalcolare questa informazione ogni volta.

I file 'kernel/fs/zone_...' descrivono le funzioni per la gestione del file system a zone.

Tabella 84.99. Funzioni per la gestione delle zone, definite nei file 'kernel/fs/zone_...'.

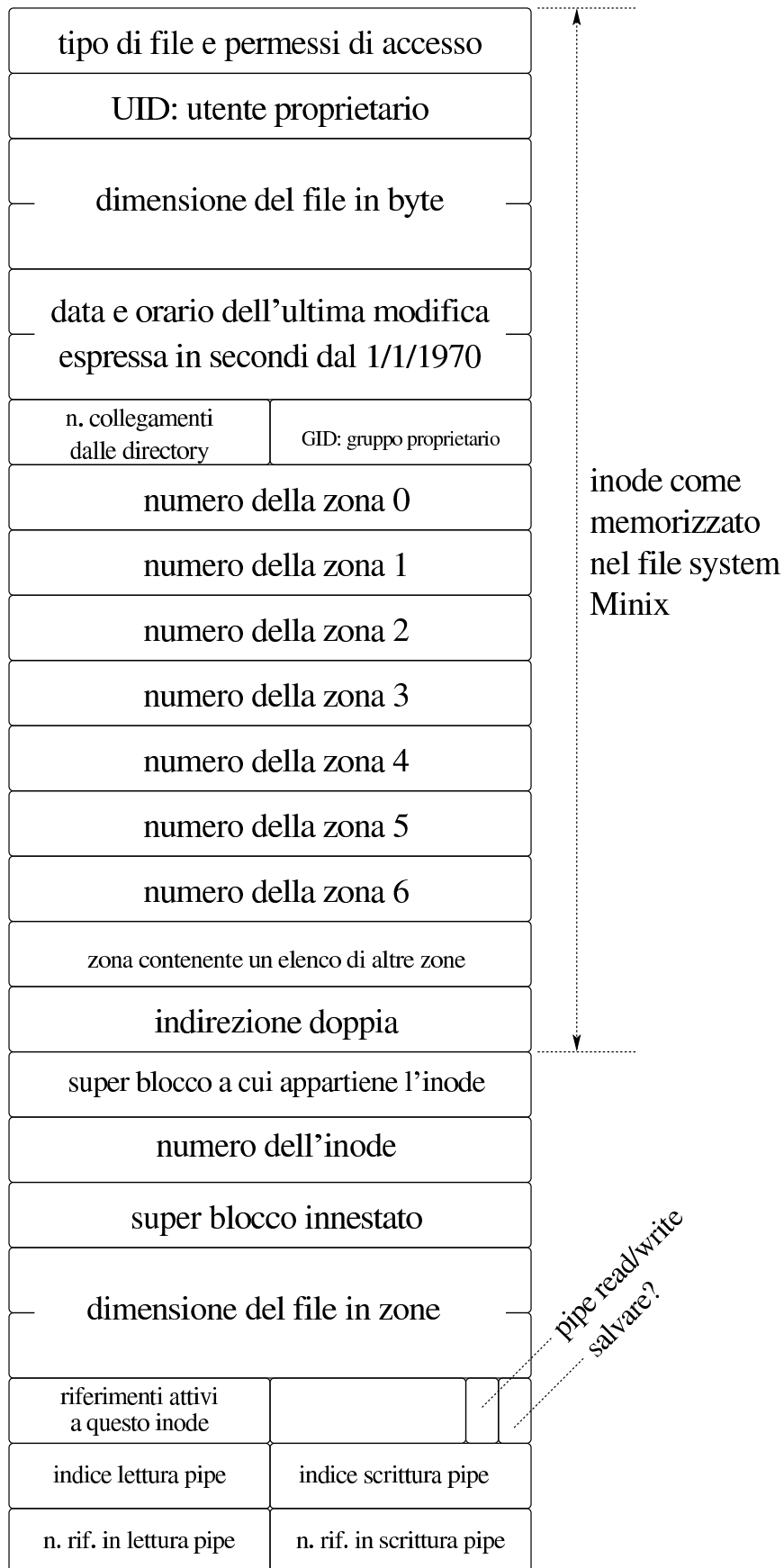
Funzione	Descrizione
<code>zno_t zone_alloc (sb_t *sb);</code>	Alloca una zona, restituendo il numero della stessa. In pratica, cerca la prima zona libera nel file system a cui si riferisce il super blocco *sb e la segna come impegnata, restituendone il numero.
<code>int zone_free (sb_t *sb, zno_t zone);</code>	Libera una zona, impegnata precedentemente.
<code>int zone_read (sb_t *sb, zno_t zone, void *buffer);</code>	Legge il contenuto di una zona, memorizzandolo a partire dalla posizione di memoria rappresentato da <i>buffer</i> .
<code>int zone_write (sb_t *sb, zno_t zone, void *buffer);</code>	Sovrascrive una zona, utilizzando il contenuto della memoria a partire dalla posizione rappresentata da <i>buffer</i> .

Funzione	Descrizione
<pre>void zone_print (sb_t *sb, zno_t zone);</pre>	Funzione diagnostica per la visualizzazione dello stato di una zona.

84.8.3 File «kernel/fs/inode_...»

I file ‘kernel/fs/inode_...’ descrivono le funzioni per la gestione dei file, in forma di inode. In uno di questi file viene dichiarata la tabella degli inode in uso nel sistema, rappresentata dall’array *inode_table[]* e per individuare un certo elemento dell’array si usa preferibilmente la funzione *inode_reference()*. Gli elementi della tabella degli inode sono di tipo ‘*inode_t*’ (definito nel file ‘kernel/fs.h’); una voce della tabella rappresenta un inode utilizzato se il campo dei riferimenti (*references*) ha un valore maggiore di zero.

Figura 84.100. Struttura del tipo 'inode_t', corrispondente agli elementi dell'array *inode_table[]*.



Listato 84.101. Struttura del tipo `'inode_t'`, corrispondente agli elementi dell'array `inode_table[]`.

```
typedef struct inode      inode_t;

struct inode {
    mode_t      mode;
    uid_t       uid;
    ssize_t     size;
    time_t      time;
    uint8_t     gid;
    uint8_t     links;
    zno_t       direct[7];
    zno_t       indirect1;
    zno_t       indirect2;
    //-----
    sb_t        *sb;
    ino_t        ino;
    sb_t        *sb_attached;
    blkcnt_t    blkcnt;
    unsigned char references;
    char        changed   : 1,
                pipe_dir  : 1;
    unsigned char pipe_off_read;
    unsigned char pipe_off_write;
    unsigned char pipe_ref_read;
    unsigned char pipe_ref_write;
};
```

Figura 84.102. Collegamento tra la tabella degli inode e quella dei super blocchi.

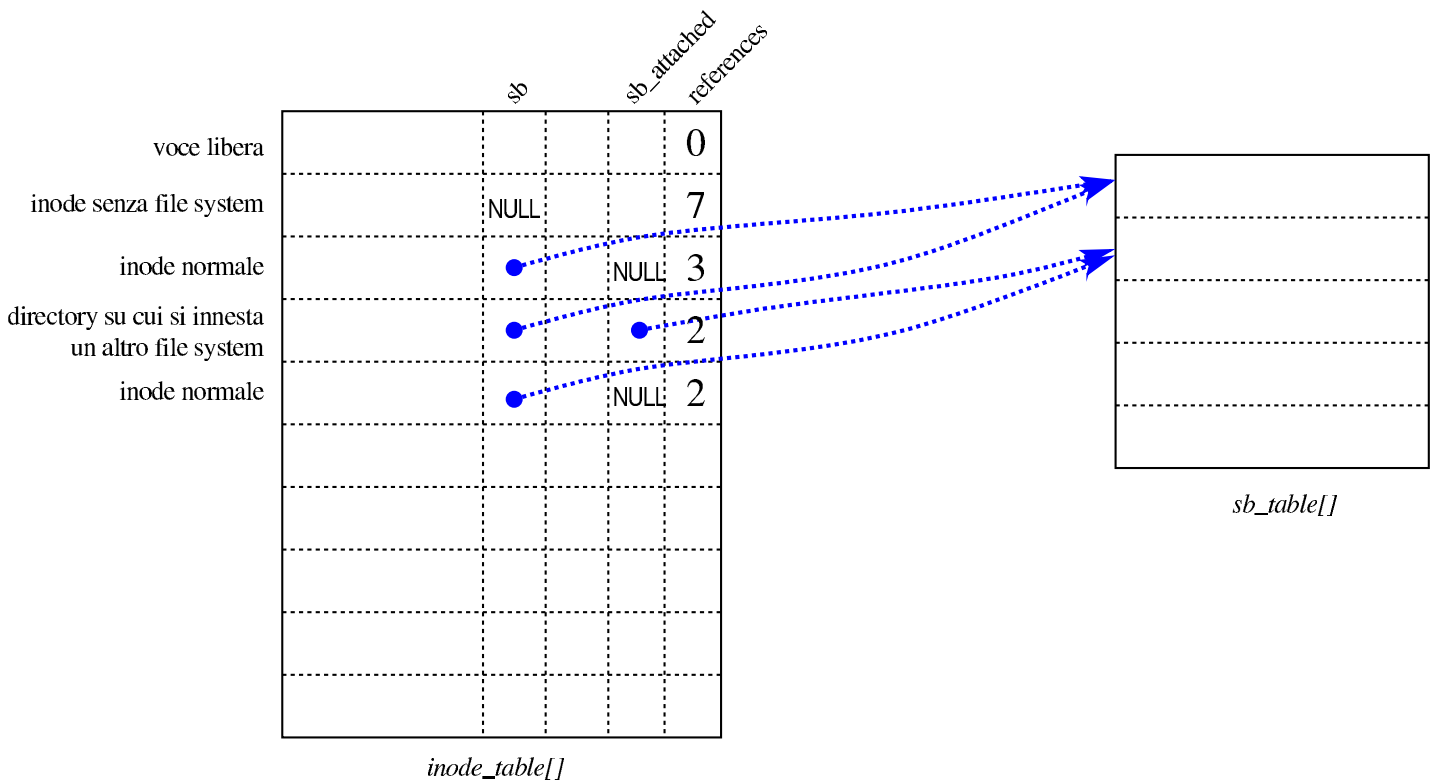


Tabella 84.103. Funzioni per la gestione dei file in forma di inode, definite nei file 'kernel/fs/inode_...'.
 inode, definite nei file 'kernel/fs/inode_...'.

Funzione	Descrizione
<pre>inode_t *inode_reference (dev_t <i>device</i>, ino_t <i>ino</i>);</pre>	<p>Restituisce il puntatore a un inode, rappresentato in pratica da un elemento dell'array <i>inode_table[]</i>, corrispondente a quello con il numero di dispositivo e di inode indicati come argomenti. Se entrambi gli argomenti sono a zero, si ottiene il puntatore al primo elemento; se entrambi i valori sono pari a -1, si ottiene il puntatore al primo elemento libero; se viene indicato il dispositivo zero e l'inode numero uno, si ottiene il puntatore all'elemento corrispondente alla directory radice del file system principale.</p>

Funzione	Descrizione
<pre>inode_t *inode_alloc (dev_t <i>device</i>, mode_t <i>mode</i>, uid_t <i>uid</i> gid_t <i>gid</i>);</pre>	<p>La funzione <i>inode_alloc()</i> cerca un inode libero nel file system del dispositivo indicato, quindi lo alloca (lo segna come utilizzato) e lo modifica aggiornando il tipo e la modalità dei permessi, oltre al proprietario del file e al gruppo. Se la funzione riesce nel suo intento, restituisce il puntatore all'inode in memoria, il quale rimane così aperto e disponibile per ulteriori elaborazioni.</p>
<pre>int inode_free (inode_t *<i>inode</i>);</pre>	<p>Segna l'inode indicato come libero.</p>

Funzione	Descrizione
<pre>inode_t *inode_get (dev_t <i>device</i>, ino_t <i>ino</i>);</pre>	<p>Restituisce il puntatore all'inode rappresentato dal numero di dispositivo e di inode, indicati come argomenti. Se l'inode è già presente nella tabella degli inode, la cosa si risolve nell'incremento di una unità del numero dei riferimenti di tale inode; se invece l'inode non è ancora presente, questo viene caricato dal suo file system nella tabella e gli viene attribuito inizialmente un riferimento attivo.</p>

Funzione	Descrizione
<pre>int inode_put (inode_t *<i>inode</i>);</pre>	Rilascia un inode che non serve più. Ciò comporta la riduzione del contatore dei riferimenti nella tabella degli inode, tenendo conto che se tale valore raggiunge lo zero, si provvede anche al suo salvataggio nel file system (ammesso che l'inode della tabella risulti modificato, rispetto alla versione presente nel file system). La funzione restituisce zero in caso di successo, oppure -1 in caso contrario.
<pre>int inode_save (inode_t *<i>inode</i>);</pre>	Salva l'inode nel file system, se questo risulta modificato.
<pre>int inode_truncate (inode_t *<i>inode</i>);</pre>	Riduce la dimensione del file a cui si riferisce l'inode a zero. In pratica fa sì che le zone allocate del file siano liberate. La funzione restituisce zero se l'operazione si conclude con successo, oppure -1 in caso di problemi.

Funzione	Descrizione
<pre>zno_t inode_zone (inode_t *<i>inode</i>, zno_t <i>fzone</i>, int <i>write</i>);</pre>	<p>Restituisce il numero di zona effettivo, corrispondente a un numero di zona relativo a un certo file di un certo inode. Se il parametro <i>write</i> è pari a zero, si intende che la zona deve esistere, quindi se questa non c'è, si ottiene semplicemente un valore pari a zero; se invece l'ultimo parametro è pari a uno, nel caso la zona cercata fosse attualmente mancante, verrebbe creata al volo nel file system.</p>

Funzione	Descrizione
<pre>inode_t * inode_stdio_dev_make (dev_t <i>device</i>, mode_t <i>mode</i>);</pre>	<p>Alloca un inode in memoria, riferito al dispositivo richiesto dal primo parametro, con i permessi del secondo parametro. Tale inode è adatto per essere utilizzato come flusso standard (standard input, standard output o standard error). Questa funzione viene usata solo da <i>file_stdio_dev_make()</i>, la quale, a sua volta, viene usata solo da <i>proc_sys_exec()</i>.</p>
<pre>blkcnt_t inode_fzones_read (inode_t *<i>inode</i>, zno_t <i>zone_start</i>, void *<i>buffer</i>, blkcnt_t <i>blkcnt</i>);</pre>	<p>Legge da un file, identificato attraverso il puntatore all'inode (della tabella di inode), una certa quantità di zone, a partire da una certa zona relativa al file, mettendo il risultato della lettura a partire dalla posizione di memoria rappresentata da un puntatore generico. La funzione restituisce la quantità di zone lette con successo.</p>

Funzione	Descrizione
<pre>blkcnt_t inode_fzones_write (inode_t *inode, zno_t zone_start, void *buffer, blkcnt_t blkcnt);</pre>	<p>Svolge il compito opposto della funzione <i>inode_fzones_read()</i> e attualmente non viene utilizzata.</p>
<pre>ssize_t inode_file_read (inode_t *inode, off_t offset, void *buffer, size_t count, int *eof);</pre>	<p>Legge il contenuto di un file, individuato da un inode già caricato nella tabella relativa, aggiornando eventualmente una variabile contenente l'indicatore di fine file. La funzione restituisce la quantità di byte letti con successo, oppure il valore -1 in caso di problemi.</p>
<pre>ssize_t inode_file_write (inode_t *inode, off_t offset, void *buffer, size_t count);</pre>	<p>Scriva una certa quantità di byte nel file individuato da un inode già caricato nella tabella relativa. La funzione restituisce la quantità di byte scritti effettivamente, oppure il valore -1 in caso di problemi.</p>

Funzione	Descrizione
<pre>int inode_check (inode_t *<i>inode</i>, mode_t <i>type</i>, int <i>perm</i>, uid_t <i>uid</i> gid_t <i>gid</i>);</pre>	Verifica che l'inode sia di un certo tipo e abbia i permessi di accesso necessari a un certo utente e gruppo. Nel parametro <i>type</i> si possono indicare più tipi validi. La funzione restituisce zero in caso di successo, ovvero di compatibilità, mentre restituisce -1 se il tipo o i permessi non sono adatti.
<pre>int inode_dir_empty (inode_t *<i>inode</i>);</pre>	Verifica se la directory a cui si riferisce l'inode è effettivamente una directory ed è vuota, nel qual caso restituisce il valore uno, altrimenti restituisce zero.
<pre>inode_t *inode_pipe_make (void);</pre>	Crea un condotto senza nome (<i>pipe</i>), restituendo il puntatore all'inode relativo.

Funzione	Descrizione
<pre> ssize_t inode_pipe_read (inode_t *inode, void *buffer, size_t count, int *eof); </pre>	<p>Legge il contenuto di un condotto, individuato da un inode già caricato nella tabella relativa, aggiornando eventualmente una variabile contenente l'indicatore di fine file. La funzione restituisce la quantità di byte letti con successo, oppure il valore -1 in caso di problemi.</p>
<pre> ssize_t inode_pipe_write (inode_t *inode, void *buffer, size_t count); </pre>	<p>Scrive una certa quantità di byte nel condotto individuato da un inode già caricato nella tabella relativa. La funzione restituisce la quantità di byte scritti effettivamente, oppure il valore -1 in caso di problemi.</p>
<pre> void inode_print (void); </pre>	<p>Funzione diagnostica per la visualizzazione sintetica del contenuto della tabella degli inode.</p>

84.8.4 Fasi dell'innesto di un file system

L'innesto e il distacco di un file system, coinvolge simultaneamente la tabella dei super blocchi e quella degli inode. Si distinguono due



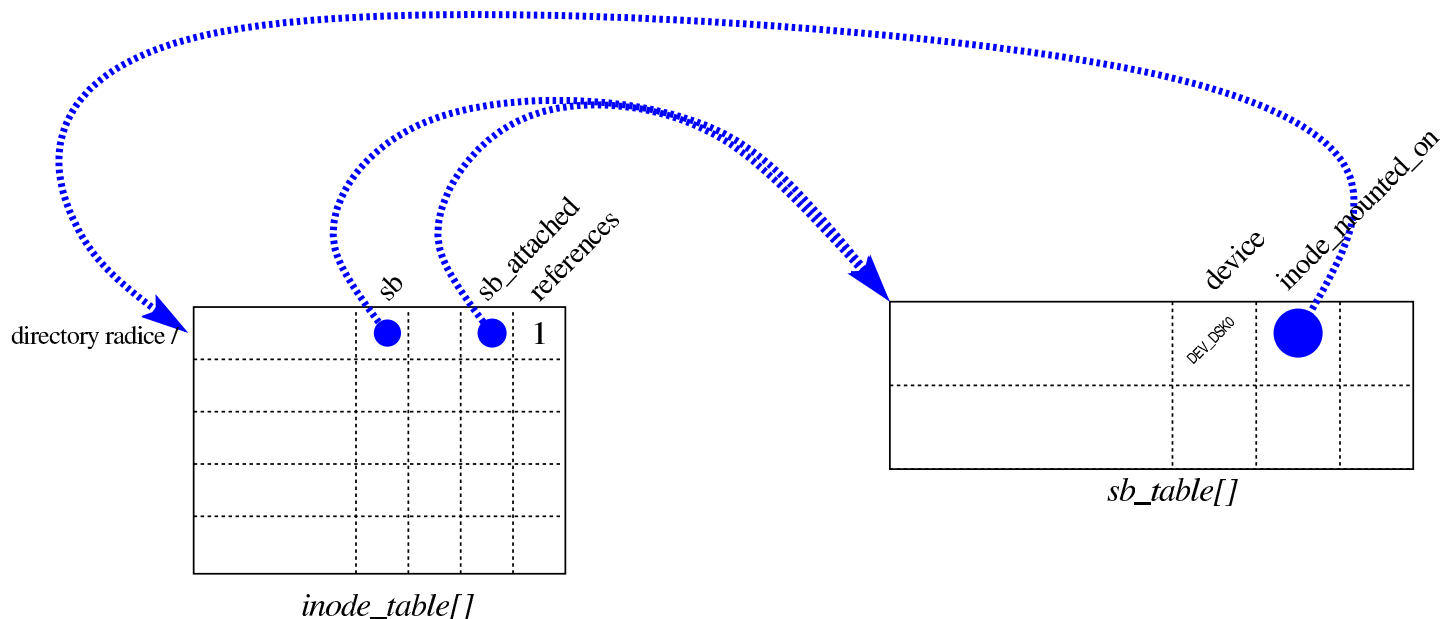
situazioni fondamentali: l'innesto del file system principale e quello di un file system ulteriore.

Quando si tratta dell'innesto del file system principale, la tabella dei super blocchi è priva di voci e quella degli inode non contiene riferimenti a file system. La funzione *sb_mount()* viene chiamata indicando, come riferimento all'inode di innesto, il puntatore a una variabile puntatore contenente il valore nullo:

```
...
inode_t *inode;
sb_t     *sb;
...
inode = NULL;
sb = sb_mount (DEV_DSK0, &inode, MOUNT_DEFAULT);
...
```

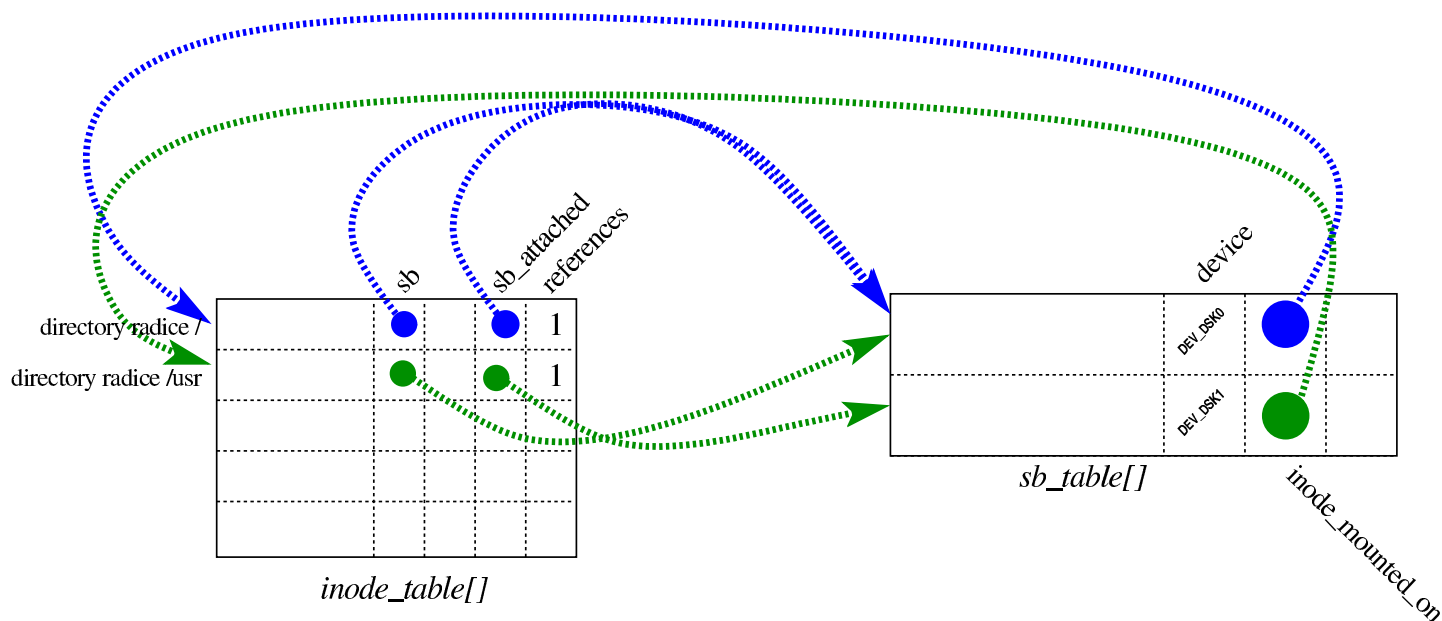
La funzione *sb_mount()* carica il super blocco nella tabella relativa, ma trovando il riferimento all'inode di innesto nullo, provvede a caricare l'inode della directory radice dello stesso dispositivo, creando un collegamento incrociato tra le tabelle dei super blocchi e degli inode, come si vede nella figura successiva.

Figura 84.105. Collegamento tra la tabella degli inode e quella dei super blocchi, quando si innesta il file system principale.



Per innestare un altro file system, occorre prima disporre dell'inode di una directory (appropriata) nella tabella degli inode, quindi si può caricare il super blocco del nuovo file system, creando il collegamento tra directory e file system innestato.

Figura 84.106. Innesto di un file system nella directory '/usr/'.

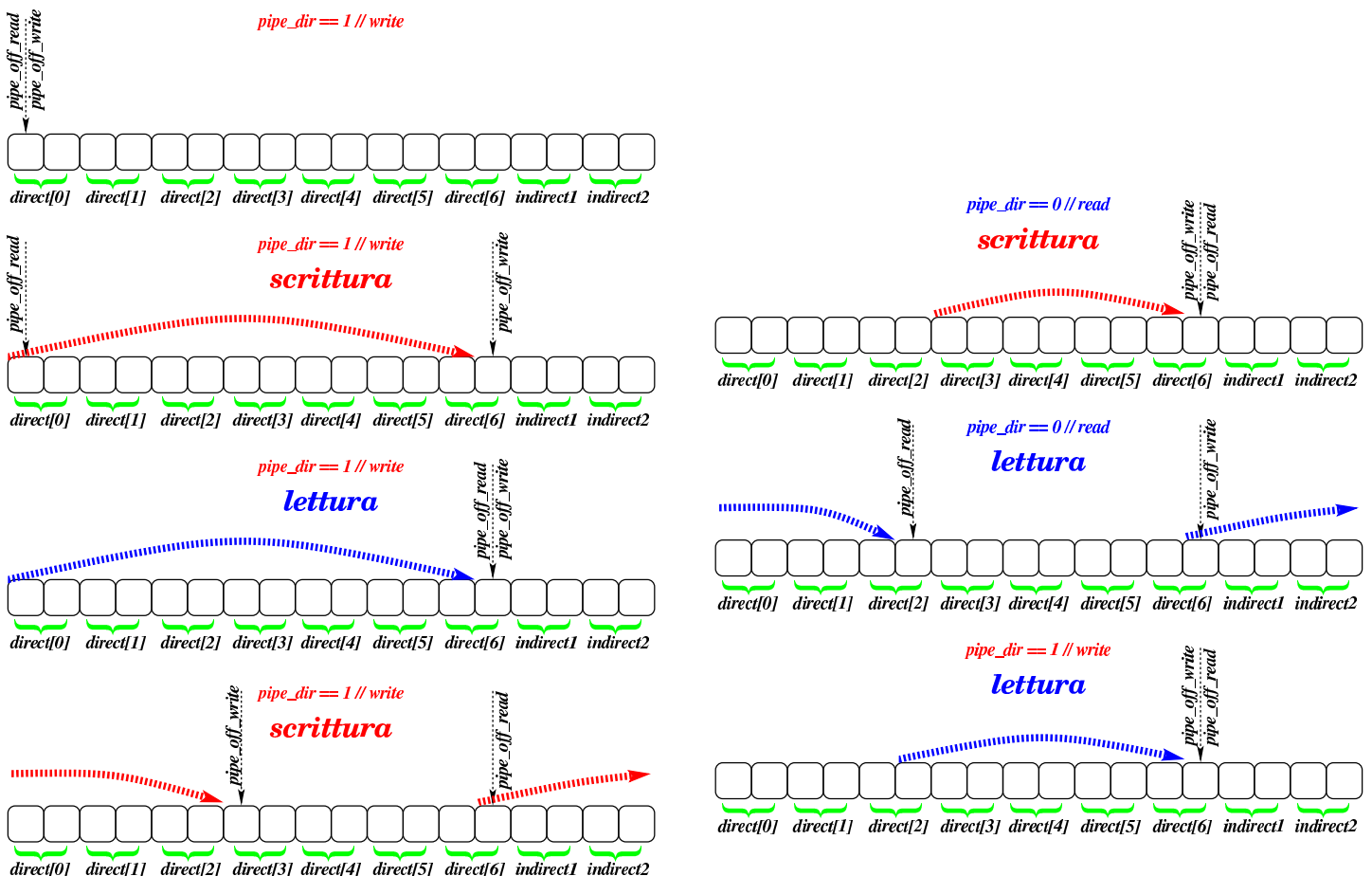


84.8.5 Condotti

I condotti sono gestiti da os32 nel modo tradizionale, sfruttando nell'inode la poca memoria che altrimenti servirebbe per i riferimenti ai blocchi di dati. In pratica, secondo la struttura del tipo `'inode_t'`, si usa `direct[]`, `indirect1` e `indirect2`. Ciò comporta complessivamente la disponibilità di soli 18 byte, cosa che comunque sarebbe insufficiente per lo standard attuale dei sistemi Unix.

Lo spazio di questi 18 byte viene trattato come una coda e scandito attraverso due indici: quello di scrittura e quello di lettura. Durante l'accesso all'inode che rappresenta un condotto si distinguono due stati, individuati dal bit `pipe_dir`.

Figura 84.107. Esempio di utilizzo di un condotto, attraverso varie fasi di scrittura e lettura.



La figura mostra un esempio che dovrebbe chiarire il meccanismo di funzionamento del condotto.

1. Inizialmente gli indici di scrittura e lettura si trovano ad avere lo stesso valore (nella figura si trovano nella posizione zero, ma qualunque altra posizione sarebbe equivalente), mentre il bit *pipe_dir* indica «scrittura». In questa situazione, si deve procedere con la scrittura, durante la quale l'indice di scrittura non può superare nuovamente quello di lettura.
2. Nel secondo disegno della figura si vede che è avvenuta una scrittura che ha occupato 13 byte, mentre l'indice di scrittura si trova sul quattordicesimo (quello successivo all'ultimo byte scritto).
3. A questo punto, si suppone che inizi la lettura del condotto: in tal caso, dato che il bit *pipe_dir* indica ancora «scrittura», la lettura non può superare la posizione che ha raggiunto l'indice di scrittura. Pertanto si suppone che la lettura raccolga la stessa quantità di byte occupati precedentemente dalla scrittura. Quando l'indice di lettura incontra quello di scrittura, il bit *pipe_dir* deve essere impostato a «scrittura», perché non c'è altro che si possa leggere. Tale bit era già impostato nel modo corretto e quindi non si notano variazioni.
4. Nel quarto disegno si vede l'inizio di una nuova fase di scrittura che raggiunge la fine dello spazio dei 18 byte previsti per riprendere dall'inizio. In questa fase di scrittura gli indici non si incontrano e nulla cambia nello stato di *pipe_dir*.
5. Nel quinto disegno (all'inizio del lato destro), la scrittura riprende e raggiunge l'indice di lettura (che non può essere superato). Qui

lo stato rappresentato da *pipe_dir* cambia, dal momento che non si può più procedere con la scrittura, adesso indica «lettura».

6. Nel sesto disegno si vede l'inizio di una lettura. Dopo di questa lettura, potrebbe esserci una fase di scrittura, ma senza poter superare l'indice di lettura. Comunque, questa scrittura non viene eseguita.
7. Nell'ultimo disegno si vede che la lettura continua, fino a raggiungere l'indice di scrittura, quando così il valore di *pipe_dir* viene invertito nuovamente.

In pratica, quando gli indici di lettura e scrittura coincidono, per sapere se si può procedere con una scrittura o una lettura, occorre chiederlo a *pipe_dir*; diversamente, con indici diversi, la scrittura o la lettura può procedere indifferentemente, ma solo fino al raggiungimento dell'altro indice. Poi, se è l'indice di lettura che ha appena raggiunto quello di scrittura, *pipe_dir* deve essere impostato per richiedere la scrittura; al contrario, quando è l'indice di scrittura che raggiunge quello di lettura, *pipe_dir* deve richiedere la lettura successiva.

84.8.6 File «kernel/fs/file_...»

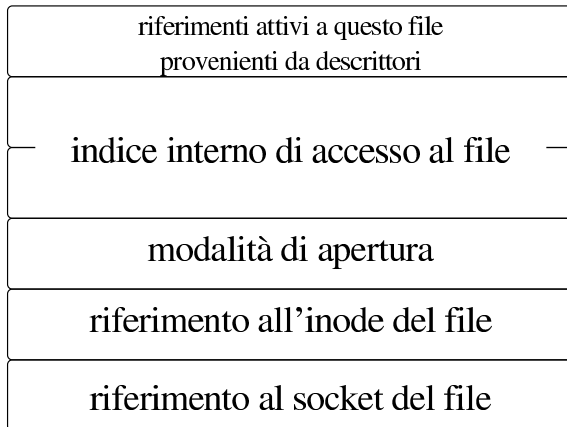
«

I file 'kernel/fs/file_...' descrivono le funzioni per la gestione della tabella dei file, la quale si collega a sua volta a quella degli inode. In realtà, le funzioni di questo gruppo sono in numero molto limitato, perché l'intervento nella tabella dei file avviene prevalentemente per opera di funzioni che gestiscono i descrittori.

La tabella dei file è rappresentata dall'array *file_table[]* e per individuare un certo elemento dell'array si usa preferibilmente la fun-

zione *file_reference()*. Gli elementi della tabella dei file sono di tipo `'file_t'` (definito nel file `'kernel/fs.h'`); una voce della tabella rappresenta un file aperto se il campo dei riferimenti (*references*) ha un valore maggiore di zero.

Figura 84.108. Struttura del tipo `'file_t'`, corrispondente agli elementi dell'array *file_table[]*.



```
typedef struct file file_t;

struct file {
    int         references;
    off_t      offset;
    int         oflags;
    inode_t    *inode;
    sock_t     *sock;
};
```

Nel membro *oflags* si annotano esclusivamente opzioni relative alla modalità di apertura del file: lettura, scrittura o entrambe; pertanto si possono usare le macro-variabili *O_RDONLY*, *O_WRONLY* e *O_RDWR*, come dichiarato nel file di intestazione `'lib/fcntl.h'`. Il membro *offset* rappresenta l'indice interno di accesso al file, per l'operazione successiva di lettura o scrittura al suo interno. Il membro *references* è un contatore dei riferimenti a questa tabella, da parte di descrittori di file.

La tabella dei file si collega a quella degli inode, attraverso il membro *inode*, oppure a quella dei socket, attraverso il membro *sock*. Più voci della tabella dei file possono riferirsi allo stesso inode (o allo stesso socket), perché hanno modalità di accesso differenti, oppure soltanto per poter distinguere l'indice interno di lettura e scrittura. Va osservato che le voci della tabella di inode potrebbero essere usa-

te direttamente e non avere elementi corrispondenti nella tabella dei file.

Figura 84.109. Collegamento tra la tabella dei file e quella degli inode.

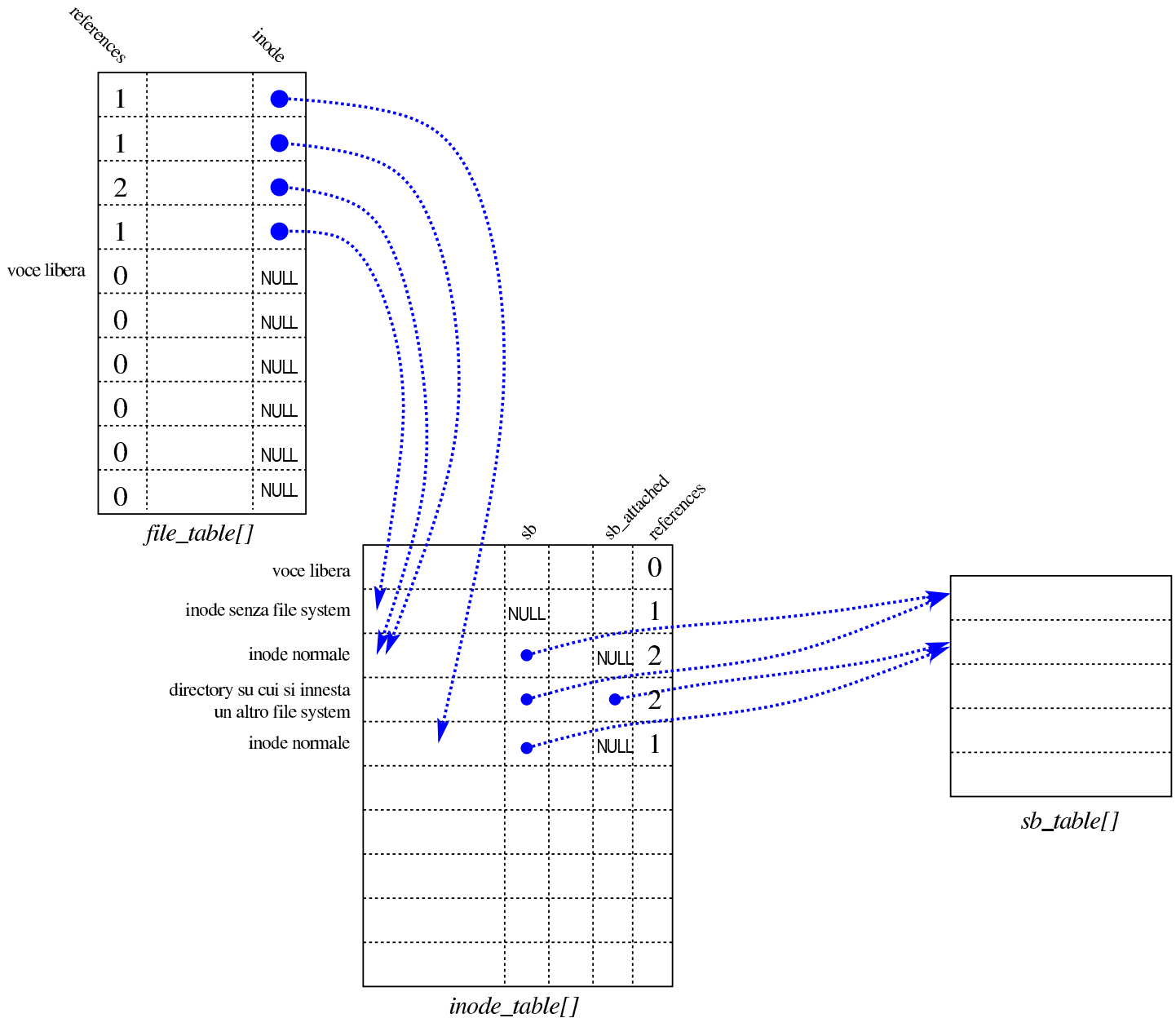


Tabella 84.110. Funzioni fatte esclusivamente per la gestione della tabella dei file *file_table[]*.

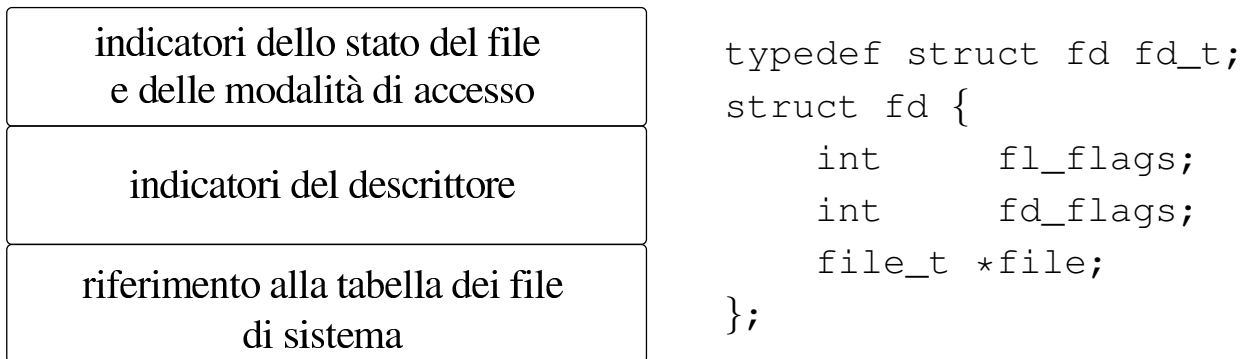
Funzione	Descrizione
<pre>file_t *file_reference (int <i>fno</i>);</pre>	Restituisce il puntatore all'elemento <i>fno</i> -esimo della tabella dei file. Se <i>fno</i> è un valore negativo, viene restituito il puntatore a una voce libera della tabella.
<pre>file_t *file_stdio_dev_make (dev_t <i>device</i>, mode_t <i>mode</i>, int <i>oflags</i>);</pre>	Crea una voce per l'accesso a un file di dispositivo standard di input-output, restituendo il puntatore alla voce stessa.

84.8.7 Descrittori di file

Le tabelle di super blocchi, inode e file, riguardano il sistema nel complesso. Tuttavia, l'accesso normale ai file avviene attraverso il concetto di «descrittore», il quale è un file aperto da un certo processo elaborativo. Nel file 'kernel/fs.h' si trova la dichiarazione e descrizione del tipo derivato '**fd_t**', usato per costruire una tabella di descrittori, ma tale tabella non fa parte della gestione del file system, bensì è incorporata nella tabella dei processi elaborativi. Pertanto, ogni processo ha una propria tabella di descrittori di file.



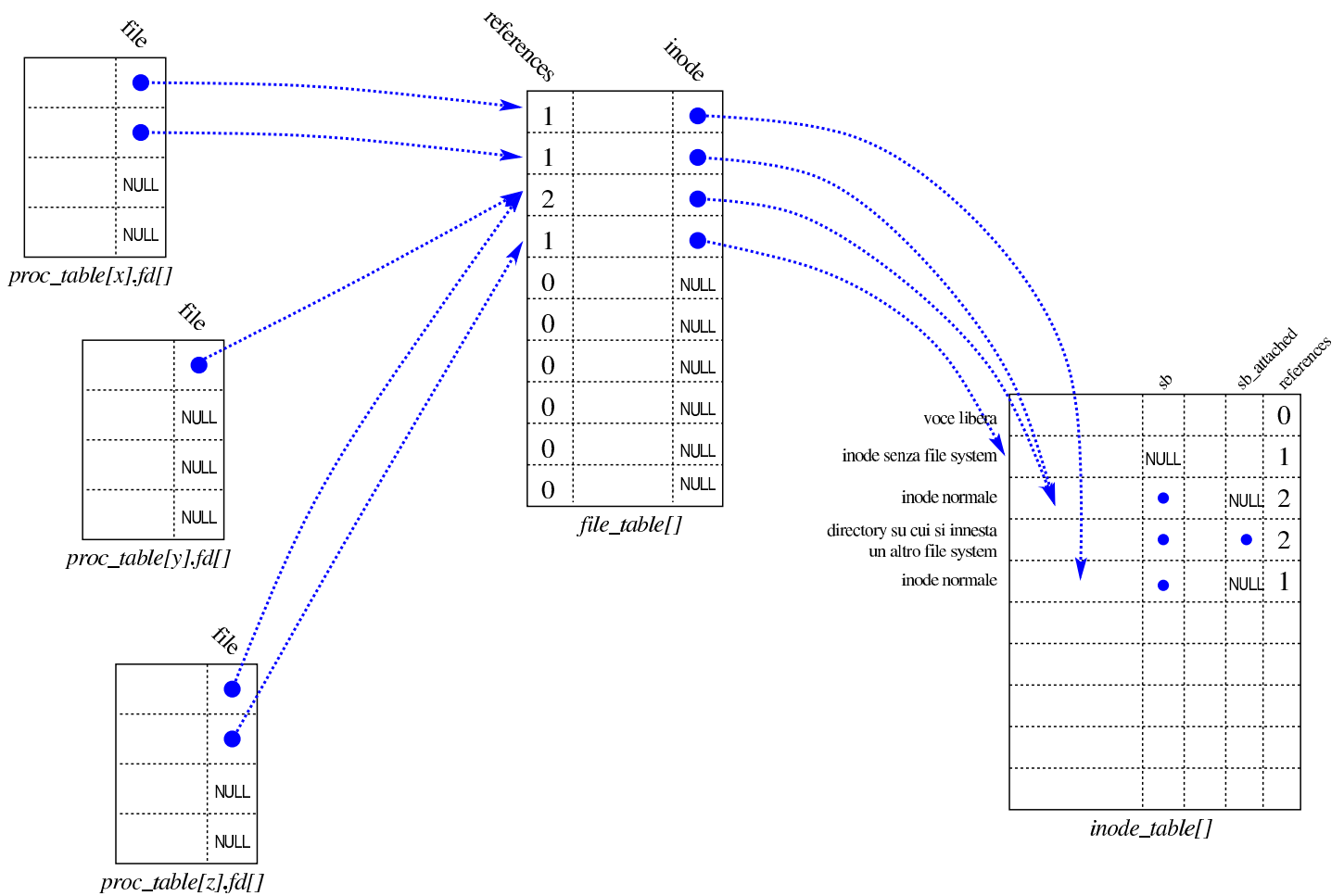
Figura 84.111. Struttura del tipo ‘**fd_t**’, con cui si costituiscono gli elementi delle tabelle dei descrittori di file, una per ogni processo.



Il membro *fl_flags* consente di annotare indicatori del tipo ‘**O_RDONLY**’, ‘**O_WRONLY**’, ‘**O_RDWR**’, ‘**O_CREAT**’, ‘**O_EXCL**’, ‘**O_NOCTTY**’, ‘**O_TRUNC**’ e ‘**O_APPEND**’, come dichiarato nella libreria standard, nel file di intestazione ‘`lib/fcntl.h`’. Tali indicatori si combinano assieme con l’operatore binario **OR**. Altri tipi di opzione che sarebbero previsti nel file ‘`lib/fcntl.h`’, sono privi di effetto nella gestione del file system di os16.

Il membro *fd_flags* serve a contenere, eventualmente, l’opzione ‘**FD_CLOEXEC**’, definita nel file ‘`lib/fcntl.h`’. Non sono previste altre opzioni di questo tipo.

Figura 84.112. Collegamento tra le tabelle dei descrittori e la tabella complessiva dei file. La tabella *proc_table[x].fd[]* rappresenta i descrittori di file del processo elaborativo *x*.



84.8.8 File «kernel/fs/path_...»

I file 'kernel/fs/path_...' descrivono le funzioni che fanno riferimento a file o directory attraverso una stringa che ne descrive il percorso.

Tabella 84.113. Funzioni per la gestione dei file, a cui si fa riferimento attraverso un percorso, senza indicazioni sul processo elaborativo.

Funzione	Descrizione
<pre>int path_fix (char *<i>path</i>);</pre>	<p>Verifica il percorso indicato semplificandolo, quindi sovrascrive il percorso originario con quello ridotto e corretto. Un percorso assoluto rimane assoluto; un percorso relativo rimane relativo, mancando qualunque indicazione sulla directory corrente.</p>
<pre>int path_full (const char *<i>path</i>, const char *<i>path_cwd</i>, char *<i>full_path</i>);</pre>	<p>Ricostruisce un percorso assoluto, usando come riferimento la directory corrente indicata in <i>path_cwd</i>, salvandolo in <i>path_full</i>.</p>

Tabella 84.114. Funzioni per la gestione dei file, a cui si fa riferimento attraverso un percorso, tenendo conto del processo elaborativo per conto del quale si svolge l'operazione. Del processo elaborativo si considera soprattutto l'identità efficace, per conoscerne i privilegi e determinare se è data effettivamente la facoltà di eseguire l'azione richiesta.

Funzione	Descrizione
<pre>inode_t *path_inode (pid_t <i>pid</i>, const char *<i>path</i>);</pre>	<p>Apre l'inode del file indicato tramite il percorso, purché il processo <i>pid</i> abbia i permessi di accesso («x») alle directory che vi conducono. La funzione restituisce il puntatore all'inode aperto, oppure il puntatore nullo se non può eseguire l'operazione.</p>
<pre>dev_t path_device (pid_t <i>pid</i>, const char *<i>path</i>);</pre>	<p>Restituisce il numero del dispositivo di un file di dispositivo; pertanto, il percorso deve fare riferimento a un file di dispositivo, per poter ottenere un risultato valido.</p>

Funzione	Descrizione
<pre>inode_t *path_inode_link (pid_t <i>pid</i>, const char *<i>path</i>, inode_t *<i>inode</i>, mode_t <i>mode</i>);</pre>	<p>Crea un collegamento fisico con il nome fornito in <i>path</i>, riferito all'inode a cui punta <i>inode</i>, ma se <i>inode</i> fosse un puntatore nullo, verrebbe semplicemente creato un file vuoto con un nuovo inode. Si richiede inoltre che il processo <i>pid</i> abbia i permessi di accesso per tutte le directory che portano al file da collegare e che nell'ultima ci sia anche il permesso di scrittura, dovendo intervenire su tale directory in questo modo. Se la funzione riesce nel proprio intento, restituisce il puntatore a ciò che descrive l'inode collegato o creato.</p>

Delle funzioni che, per affinità, farebbero parte di questo gruppo, si trovano nella directory 'kernel/lib_s/', in quanto servono per attuare delle chiamate di sistema.

Tabella 84.115. Funzioni per la gestione dei file, a cui si fa riferimento attraverso un percorso, tenendo conto del processo elaborativo per conto del quale si svolge l'operazione, contenute nella directory 'kernel/lib_s/'.

Funzione	Descrizione
<pre>int s_chdir (pid_t <i>pid</i>, const char *<i>path</i>);</pre>	<p>Cambia la directory corrente, utilizzando il nuovo percorso indicato. È l'equivalente della funzione standard <i>chdir()</i> (sezione 87.6).</p>
<pre>int s_chmod (pid_t <i>pid</i>, const char *<i>path</i>, mode_t <i>mode</i>);</pre>	<p>Cambia la modalità di accesso al file indicato. È l'equivalente della funzione standard <i>chmod()</i> (sezione 87.7).</p>
<pre>int s_chown (pid_t <i>pid</i>, const char *<i>path</i>, uid_t <i>uid</i>, gid_t <i>gid</i>);</pre>	<p>Cambia l'utente e il gruppo proprietari del file. È l'equivalente della funzione standard <i>chown()</i> (sezione 87.8).</p>
<pre>int s_link (pid_t <i>pid</i>, const char *<i>path_old</i>, const char *<i>path_new</i>);</pre>	<p>Crea un collegamento fisico. È l'equivalente della funzione standard <i>link()</i> (sezione 87.30).</p>
<pre>int s_mkdir (pid_t <i>pid</i>, const char *<i>path</i>, mode_t <i>mode</i>);</pre>	<p>Crea una directory, con la modalità dei permessi indicata. È l'equivalente della funzione standard <i>mkdir()</i> (sezione 87.34).</p>

Funzione	Descrizione
<pre>int s_mknod (pid_t <i>pid</i>, const char *<i>path</i>, mode_t <i>mode</i>, dev_t <i>device</i>);</pre>	<p>Crea un file vuoto, con il tipo e i permessi specificati da <i>mode</i>; se si tratta di un file di dispositivo, viene preso in considerazione anche il parametro <i>device</i>, per specificare il numero primario e secondario dello stesso. Va osservato che con questa funzione è possibile creare una directory priva delle voci ‘.’ e ‘..’. È l’equivalente della funzione standard <i>mknod()</i> (sezione 87.35).</p>
<pre>int s_mount (pid_t <i>pid</i>, const char *<i>path_dev</i>, const char *<i>path_mnt</i>, int <i>options</i>);</pre>	<p>Innesta il dispositivo corrispondente a <i>path_dev</i>, nella directory <i>path_mnt</i> (tenendo conto della directory corrente del processo <i>pid</i>), con le opzioni specificate. Le opzioni disponibili sono solo ‘MOUNT_DEFAULT’ e ‘MOUNT_RO’, come dichiarato nel file di intestazione ‘lib/sys/os32.h’.</p>

Funzione	Descrizione
<pre>int s_open (pid_t <i>pid</i>, const char *<i>path</i>, int <i>oflags</i>, mode_t <i>mode</i>);</pre>	<p>Apre un descrittore, fornendo però il percorso del file. È l'equivalente della funzione standard <i>open()</i> (sezione 87.37).</p>
<pre>int s_stat (pid_t <i>pid</i>, const char *<i>path</i>, struct stat *<i>buffer</i>);</pre>	<p>Aggiorna la variabile strutturata a cui punta <i>buffer</i>, con le informazioni sul file specificato. È l'equivalente della funzione standard <i>stat()</i> (sezione 87.55).</p>
<pre>int s_umount (pid_t <i>pid</i>, const char *<i>path_mnt</i>);</pre>	<p>Stacca l'unità innestata nella directory indicata, purché nulla al suo interno sia attualmente in uso.</p>
<pre>int s_unlink (pid_t <i>pid</i>, const char *<i>path</i>);</pre>	<p>Cancella un file o una directory, purché questa sia vuota. È l'equivalente della funzione standard <i>unlink()</i> (sezione 87.62).</p>

84.8.9 File «kernel/fs/fd_...»

I file 'kernel/fs/fd_...' descrivono le funzioni che fanno riferimento a file o directory attraverso il numero di descrittore, riferito a sua volta a un certo processo elaborativo. Pertanto, il numero del processo e il numero del descrittore sono i primi due parametri obbligatori di tutte queste funzioni.



Tabella 84.116. Funzioni per la gestione dei file, a cui si fa riferimento attraverso il descrittore, relativamente a un certo processo elaborativo, le quali non rappresentano direttamente la realizzazione di una chiamata di sistema.

Funzione	Descrizione
<pre>fd_t *fd_reference (pid_t <i>pid</i>, int *<i>fdn</i>);</pre>	Produce il puntatore ai dati del descrittore <i>*fdn</i> . Se <i>*fdn</i> è minore di zero, si ottiene il riferimento al primo descrittore libero, aggiornando anche <i>*fdn</i> stesso.
<pre>int fd_dup (pid_t <i>pid</i>, int <i>fdn_old</i>, int <i>fdn_min</i>);</pre>	Duplica il descrittore <i>fdn_old</i> , creandone un altro con numero maggiore o uguale a <i>fdn_min</i> (viene scelto il primo libero a partire da <i>fdn_num</i>).

Delle funzioni che, per affinità, farebbero parte di questo gruppo, si trovano nella directory ‘kernel/lib_s/’, in quanto servono per attuare delle chiamate di sistema.

Tabella 84.117. Funzioni per la gestione dei file, a cui si fa riferimento attraverso il descrittore, tenendo conto del processo elaborativo per conto del quale si svolge l’operazione, contenute nella directory ‘kernel/lib_s/’.

Funzione	Descrizione
<pre>int s_dup (pid_t <i>pid</i>, int <i>fdn_old</i>);</pre>	<p>Duplica il descrittore <i>fdn_old</i>, creandone un altro (utilizzando il primo numero di descrittore libero) il cui numero viene restituito dalla funzione. È l'equivalente della funzione standard <i>dup()</i> (sezione 87.12).</p>
<pre>int s_dup2 (pid_t <i>pid</i>, int <i>fdn_old</i>, int <i>fdn_new</i>);</pre>	<p>Duplica il descrittore <i>s_old</i>, creandone un altro con numero <i>fdn_new</i>. Se però <i>fdn_new</i> è già aperto, prima della duplicazione questo viene chiuso. È l'equivalente della funzione standard <i>dup2()</i> (sezione 87.12).</p>
<pre>int s_fchmod (pid_t <i>pid</i>, int <i>fdn</i>, mode_t <i>mode</i>);</pre>	<p>Cambia la modalità dei permessi (solo gli ultimi 12 bit del parametro <i>mode</i> vengono considerati). È l'equivalente della funzione standard <i>fchmod()</i> (sezione 87.7).</p>
<pre>int s_fchown (pid_t <i>pid</i>, int <i>fdn</i>, uid_t <i>uid</i>, gid_t <i>gid</i>);</pre>	<p>Cambia la proprietà (utente e gruppo). È l'equivalente della funzione standard <i>fchown()</i> (sezione 87.8).</p>

Funzione	Descrizione
<pre>int s_fcntl (pid_t <i>pid</i>, int <i>fdn</i>, int <i>cmd</i>, int <i>arg</i>);</pre>	<p>Svolge il compito della funzione standard <i>fcntl()</i> (sezione 87.18).</p>
<pre>int s_fstat (pid_t <i>pid</i>, int <i>fdn</i>, struct stat *<i>buffer</i>);</pre>	<p>Svolge il compito della funzione standard <i>fstat()</i> (sezione 87.55).</p>
<pre>off_t s_lseek (pid_t <i>pid</i>, int <i>fdn</i>, off_t <i>offset</i>, int <i>whence</i>);</pre>	<p>Riposiziona l'indice interno di accesso del descrittore di file. È l'equivalente della funzione standard <i>lseek()</i> (sezione 87.33).</p>
<pre>int s_pipe (pid_t <i>pid</i>, int <i>pipefd</i>[2]);</pre>	<p>Crea un condotto senza nome (<i>pipe</i>), per il quale, i due descrittori necessari vengono salvati in <i>pipefd</i>[]. Per la precisione, <i>pipefd</i>[0] individua il descrittore del lato di lettura, mentre <i>pipefd</i>[1] individua quello del lato di scrittura. È l'equivalente della funzione standard <i>pipe()</i> (sezione 87.38).</p>

Funzione	Descrizione
<pre> ssize_t s_read (pid_t <i>pid</i>, int <i>fdn</i>, void *<i>buffer</i>, size_t <i>count</i>, int *<i>eof</i>); </pre>	<p>Legge da un descrittore, aggiornando eventualmente la variabile <i>*eof</i> in caso di fine del file. È l'equivalente della funzione standard <i>read()</i> (sezione 87.39).</p>
<pre> ssize_t s_write (pid_t <i>pid</i>, int <i>fdn</i>, const void *<i>buffer</i>, size_t <i>count</i>); </pre>	<p>Scrive nel descrittore. È l'equivalente della funzione standard <i>write()</i> (sezione 87.64).</p>

84.9 Gestione delle interfacce di rete

Il sistema os32 può gestire soltanto interfacce di rete Ethernet e l'interfaccia virtuale locale, nota con il nome *loopback*. Le interfacce di rete hanno tutte nomi del tipo '**net***n*', dove *n* è un numero intero, a partire da zero, e di norma l'interfaccia virtuale locale coincide con il nome '**net0**'.

84.9.1 Gestione dei dispositivi NE2K

Il kernel di os32 è in grado di gestire soltanto le interfacce di rete Ethernet NE2000, collocate nel bus PCI: NE2K. Ciò consente di conoscere la porta di I/O necessaria per accedervi, in modo automatico. Le funzioni per la gestione di queste interfacce sono contenute nei file della directory '`kernel/driver/nic/ne2k/`' e fanno capo al file di intestazione '`kernel/driver/nic/ne2k.h`' (listato [94.4.19](#) e successivi).

Le interfacce di rete NE2000 dispongono di una piccola memoria tampone interna per la ricezione; tuttavia, appena viene individuato un pacchetto ricevuto, os32 lo trasferisce immediatamente in una propria memoria tampone, contenuta nella tabella delle interfacce, descritta nella sezione successiva.

Per una maggiore semplicità progettuale, la trasmissione di un pacchetto avviene mettendo tutto il sistema in attesa, fino a che l'interfaccia dà un responso, positivo o negativo che sia. Tuttavia, ciò comporta anche il rischio di bloccare definitivamente il sistema, nel caso si dovessero manifestare dei problemi all'interfaccia.

Tabella 84.118. Funzioni per la gestione dell'interfaccia di rete NE2000.

Funzione	Descrizione
<pre>int ne2k_check (uintptr_t io);</pre>	Verifica se l'interfaccia corrispondente alla porta di I/O specificata è veramente di tipo NE2000 [94.4.20].
<pre>int ne2k_isr (uintptr_t io);</pre>	Verifica lo stato dell'interfaccia e ne acquisisce i dati, se disponibili. In caso di ricezione di una trama, viene chiamata la funzione <i>ne2k_rx()</i> per trasferirla nella memoria tampone della tabella delle interfacce [94.4.21].

Funzione	Descrizione
<pre>int ne2k_isr_expect (uintptr_t <i>io</i>, unsigned int <i>isr_expect</i>);</pre>	<p>Rimane in attesa fino a che il registro ISR dell'interfaccia si attiva almeno un indicatore corrispondente a quanto richiesto con il parametro <i>isr_expect</i>. Questa funzione viene usata, in particolare, nella trasmissione dei pacchetti, per i quali occorre verificare quando l'interfaccia ha completato il procedimento [94.4.22].</p>
<pre>int ne2k_reset (uintptr_t <i>io</i>, void *<i>address</i>);</pre>	<p>Azzerare l'interfaccia e ne estrae l'indirizzo fisico, collocandolo in corrispondenza di <i>*address</i> [94.4.23].</p>
<pre>int ne2k_rx (uintptr_t <i>io</i>);</pre>	<p>Copia tutte le trame accumulate nella memoria tampone interna dell'interfaccia, in quella della tabella delle interfacce [94.4.24].</p>
<pre>int ne2k_rx_reset (uintptr_t <i>io</i>);</pre>	<p>Reinizializza il processo di ricezione [94.4.25].</p>

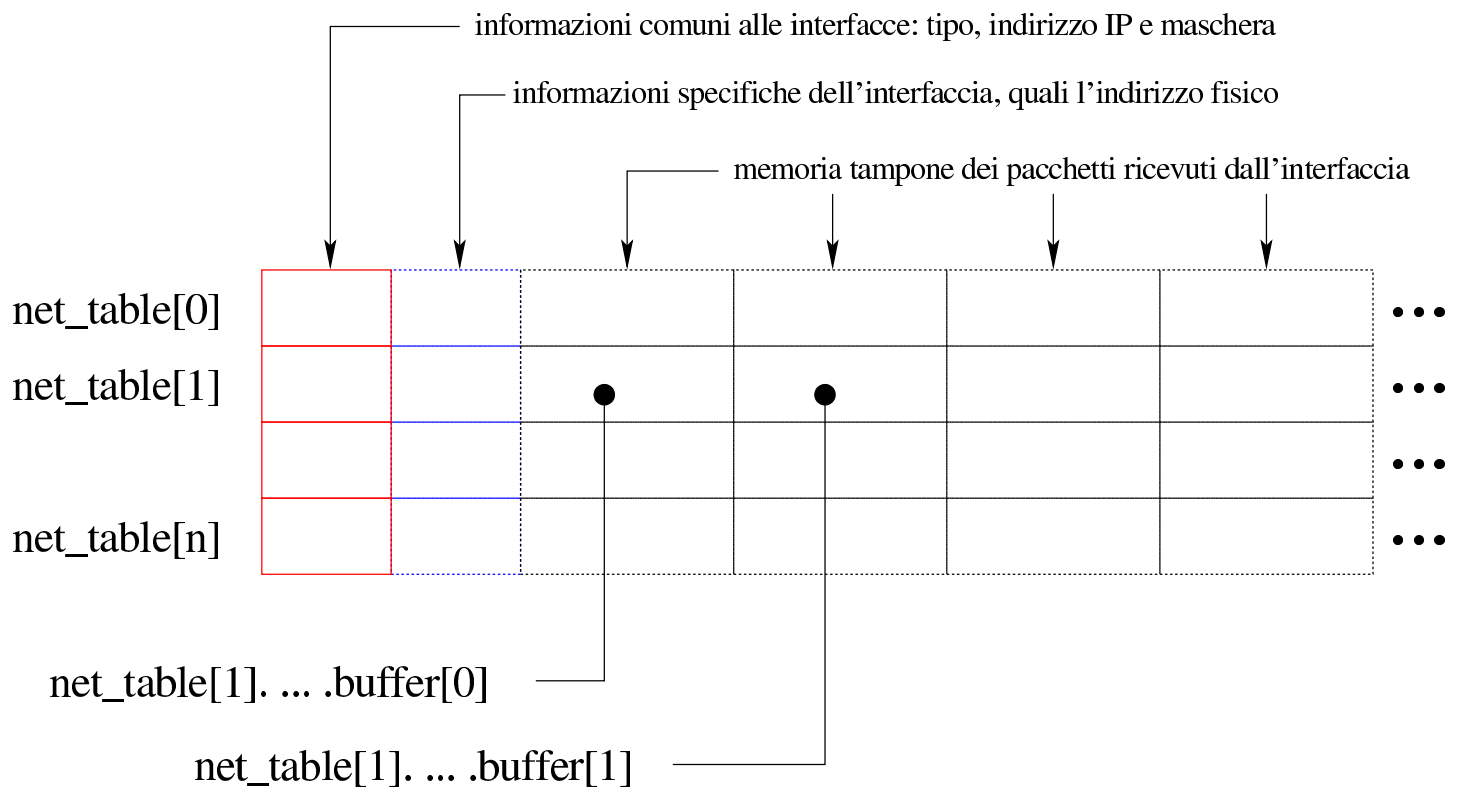
Funzione	Descrizione
<pre data-bbox="108 369 943 482">int ne2k_tx (uintptr_t <i>io</i>, void *<i>buffer</i>, size_t <i>size</i>);</pre>	<p data-bbox="970 159 1487 656">Trasmette una trama Ethernet, contenuta all'interno di <i>*buffer</i>, della lunghezza specificata da <i>size</i>. La funzione attende il completamento dell'operazione, prima di concludere il proprio funzionamento [94.4.26].</p>

84.9.2 Tabella delle interfacce e funzioni accessorie

«

Nei file 'kernel/net/net_public.c' [94.12.31] e 'kernel/net.h' [94.12] viene dichiarata la tabella delle interfacce, corrispondente all'array *net_table[]*, con lo scopo di contenere la memoria tampone delle trame ricevute da ogni interfaccia. La struttura della tabella è definita dal tipo '*net_t*' e appare semplificata nella figura successiva.

Figura 84.119. Struttura semplificata della tabella delle interfacce.



Listato 84.120. Struttura di ogni elemento della tabella delle interfacce; i dettagli dei membri *buffer* non sono evidenziati, ma contengono sempre, a loro volta, i membri *clock* e *size*

```
typedef struct {
    clock_t          clock;
    size_t          size;
    ...
} net_buffer_eth_t | net_buffer_lo_t;

typedef struct {
    unsigned int type;
    h_addr_t      ip;          // IPv4 address in host byte order.
    uint8_t       m;          // Short netmask.
    union {
        //
```

```
// Ethernet type data:
//
struct {
    uint8_t          mac[6];
    uintptr_t       base_io;
    unsigned char    irq;
    net_buffer_eth_t buffer[NET_MAX_BUFFERS];
} ethernet;
//
// Loopback type data:
//
struct {
    net_buffer_lo_t buffer[NET_MAX_BUFFERS];
} loopback;
};
} net_t;
```

Ogni pacchetto accumulato nella memoria tampone della tabella delle interfacce, oltre al contenuto del pacchetto, include l'orario in cui questo è stato ricevuto (in unità 'clock_t') e la sua dimensione effettiva.

La scansione della tabella richiede generalmente due indici: il numero che individua l'interfaccia e il numero che rappresenta la trama memorizzata (PDU di livello 2 nel caso di interfaccia Ethernet, oppure di livello 3 nel caso di interfaccia virtuale locale), assieme a delle informazioni accessorie. Per esempio, 'net_table[0].loopback.buffer[f].clock' individua l'orario di ricevimento di un pacchetto con indice *f* dell'interfaccia locale 'net0' (*loopback*), mentre 'net_table[1].ethernet.buffer[f].size' individua la dimensione del pacchetto *f* dell'interfaccia Ethernet 'net1'.

I pacchetti, a livello della rete fisica, vengono depositati nella memoria tampone della tabella, in corrispondenza dell'interfaccia da cui provengono; da qui, poi, attraverso la funzione *net_rx()*, i pacchetti vengono passati ai gestori appropriati, cancellandoli dalla tabella originaria.

Tabella 84.121. Funzioni per la gestione della tabella delle interfacce, contenute nella directory 'kernel/net/', e altre accessorie relative alla gestione Ethernet.

Funzione	Descrizione
<pre>net_buffer_eth_t * net_buffer_eth (int <i>n</i>);</pre>	Restituisce il puntatore a un elemento libero, o utilizzabile, della memoria tampone dell'interfaccia ' <i>netn</i> ', purché questa sia di tipo Ethernet [94.12.23].
<pre>net_buffer_lo_t * net_buffer_lo (int <i>n</i>);</pre>	Restituisce il puntatore a un elemento libero, o utilizzabile, della memoria tampone dell'interfaccia ' <i>netn</i> ', purché questa sia di tipo <i>loopback</i> , ossia l'interfaccia locale virtuale [94.12.24].
<pre>int net_index (h_addr_t <i>ip</i>);</pre>	Restituisce l'indice della tabella delle interfacce, corrispondente all'indirizzo IPv4 fornito come argomento [94.12.27].

Funzione	Descrizione
<pre>int net_index_eth (h_addr_t <i>ip</i>, uint8_t <i>mac</i>[6], uintptr_t <i>io</i>);</pre>	Restituisce l'indice della tabella delle interfacce, corrispondente a uno dei dati forniti come argomento (i valori nulli vengono ignorati), purché si tratti di un'interfaccia Ethernet [94.12.28].
<pre>void net_init (void);</pre>	Inizializza la gestione della rete, utilizzando le informazioni attraverso le opzioni di avvio per configurare anche le interfacce Ethernet [94.12.29].
<pre>void net_rx (void);</pre>	Scandisce i pacchetti memorizzati nella tabella delle interfacce, passandoli al gestore appropriato e rimuovendoli poi dalla tabella [94.12.32].

Funzione	Descrizione
<pre data-bbox="108 425 938 658">int net_eth_ip_tx (h_addr_t <i>src</i>, h_addr_t <i>dst</i>, const void *<i>packet</i>, size_t <i>size</i>);</pre>	<p data-bbox="970 159 1485 887">A partire da un pacchetto IPv4 completo e dagli indirizzi IPv4 di origine e di destinazione, viene assemblata e spedita una trama Ethernet. La funzione richiede separatamente l'indicazione degli indirizzi IPv4 di origine e destinazione, per semplificare il codice, evitando di estrapolarli dal pacchetto IPv4 stesso [94.12.25].</p>
<pre data-bbox="108 997 737 1169">int net_eth_tx (int <i>n</i>, void *<i>buffer</i>, size_t <i>size</i>);</pre>	<p data-bbox="970 903 1485 1226">Provvede a trasmettere una trama Ethernet attraverso l'interfaccia <i>n</i> (ovvero <i>net_table[n]</i>), la quale deve essere di tipo Ethernet [94.12.26].</p>

84.9.3 Tabella ARP

Per mantenere memoria delle corrispondenze tra indirizzi IPv4 e indirizzi Ethernet, si utilizza la tabella ARP, descritta nel file 'kernel/net/arp.h' [94.12.1] e dichiarata nel file 'kernel/net/arp/arp_public.c' [94.12.6].

Le voci della tabella sono valide per un tempo limitato, definito dalla macro-variabile *ARP_MAX_TIME* e periodicamente vengono scandite e cancellate le voci troppo vecchie.

Figura 84.122. Struttura della tabella ARP, costituita da elementi di tipo ‘arp_t’.

	time	MAC	IPv4	
mac_table[0]				<pre>typedef struct { time_t time; uint8_t mac[6]; h_addr_t ip; } arp_t;</pre>
mac_table[1]				
mac_table[n]				

Tabella 84.123. Funzioni per la gestione della tabella ARP, contenute nella directory ‘kernel/net/arp/’.

Funzione	Descrizione
<pre>void arp_init (void);</pre>	Azzera completamente la tabella ARP: si usa una volta sola all’avvio della gestione della rete [94.12.4].
<pre>void arp_clean (void);</pre>	Azzera le voci della tabella ARP che risultano troppo vecchie e che devono essere rinnovate [94.12.2].
<pre>int arp_index (unsigned char <i>mac[6]</i>, h_addr_t <i>ip</i>);</pre>	Restituisce l’indice della tabella ARP, corrispondente all’indirizzo Ethernet o all’indirizzo IPv4 fornito [94.12.3].

Funzione	Descrizione
<pre>arp_t *arp_reference (void);</pre>	Restituisce il puntatore a un elemento della tabella ARP contenente la voce più vecchia, allo scopo presumibile di riutilizzarla per un indirizzo nuovo [94.12.7].
<pre>void arp_request (h_addr_t <i>ip</i>);</pre>	Invia una richiesta ARP, preparando il pacchetto relativo e inviandolo attraverso la funzione <i>ethernet_tx()</i> [94.12.8].
<pre>int arp_rx (int <i>n</i>, int <i>f</i>);</pre>	Legge dalla tabella delle interfacce il pacchetto individuato dall'indice <i>n</i> per l'interfaccia e dall'indice <i>f</i> per la trama relativa. Il pacchetto in questione deve essere relativo al protocollo ARP: se si tratta di una richiesta, provvede a inviare una risposta, se invece si tratta di una risposta, allora aggiorna la tabella ARP [94.12.9].

84.10 Gestione di IPv4



La gestione di IPv4, da parte di os32, è estremamente limitata, per semplificare il codice e la sua comprensione. In particolare non si considerano le opzioni che potrebbero essere contenute tra l'intestazione minima e il contenuto del pacchetto IPv4.

Tabella 84.124. Funzioni per la gestione dei pacchetti a livello IP.

Funzione	Descrizione
<pre>uint16_t ip_checksum (uint16_t *data1, size_t size1, uint16_t *data2, size_t size2);</pre>	<p>Produce il codice di controllo usato nel protocollo IPv4, partendo da due blocchi di dati [94.12.16].</p>
<pre>int ip_rx (int n, int f);</pre>	<p>Si occupa di acquisire un pacchetto IPv4, dalla tabella <i>net_table[]</i>, per copiarlo nella tabella <i>ip_table[]</i> e trattarlo per le questioni urgenti [94.12.21].</p>
<pre>ip_t *ip_reference (void);</pre>	<p>Restituisce il puntatore a un elemento della tabella <i>ip_table[]</i> che possa essere riutilizzato, perché mai usato prima oppure perché contenente il pacchetto IPv4 ricevuto che risulta essere più vecchio di tutti gli altri [94.12.20].</p>

Funzione	Descrizione
<pre>ssize_t ip_header (h_addr_t <i>src</i>, h_addr_t <i>dst</i>, uint16_t <i>id</i>, uint8_t <i>ttl</i>, uint8_t <i>protocol</i>, void *<i>buffer</i>, size_t <i>length</i>);</pre>	<p>Scrive, in corrispondenza di <i>buffer</i>, un'intestazione IPv4, sulla base dei dati contenuti negli altri parametri [94.12.17].</p>
<pre>int ip_tx (h_addr_t <i>src</i>, h_addr_t <i>dst</i>, int <i>protocol</i>, const void *<i>buffer</i>, size_t <i>size</i>);</pre>	<p>Produce e trasmette un pacchetto IPv4, partendo dal contenuto che deve avere e dai dati necessari a costruire l'intestazione IPv4 [94.12.22].</p>
<pre>h_addr_t ip_mask (int <i>m</i>);</pre>	<p>A partire da un numero che rappresenta la dimensione di una maschera di rete, si ottiene il valore a 32 bit della maschera stessa. Per esempio, dal valore 16, si ottiene 255.255.0.0 [94.12.18].</p>

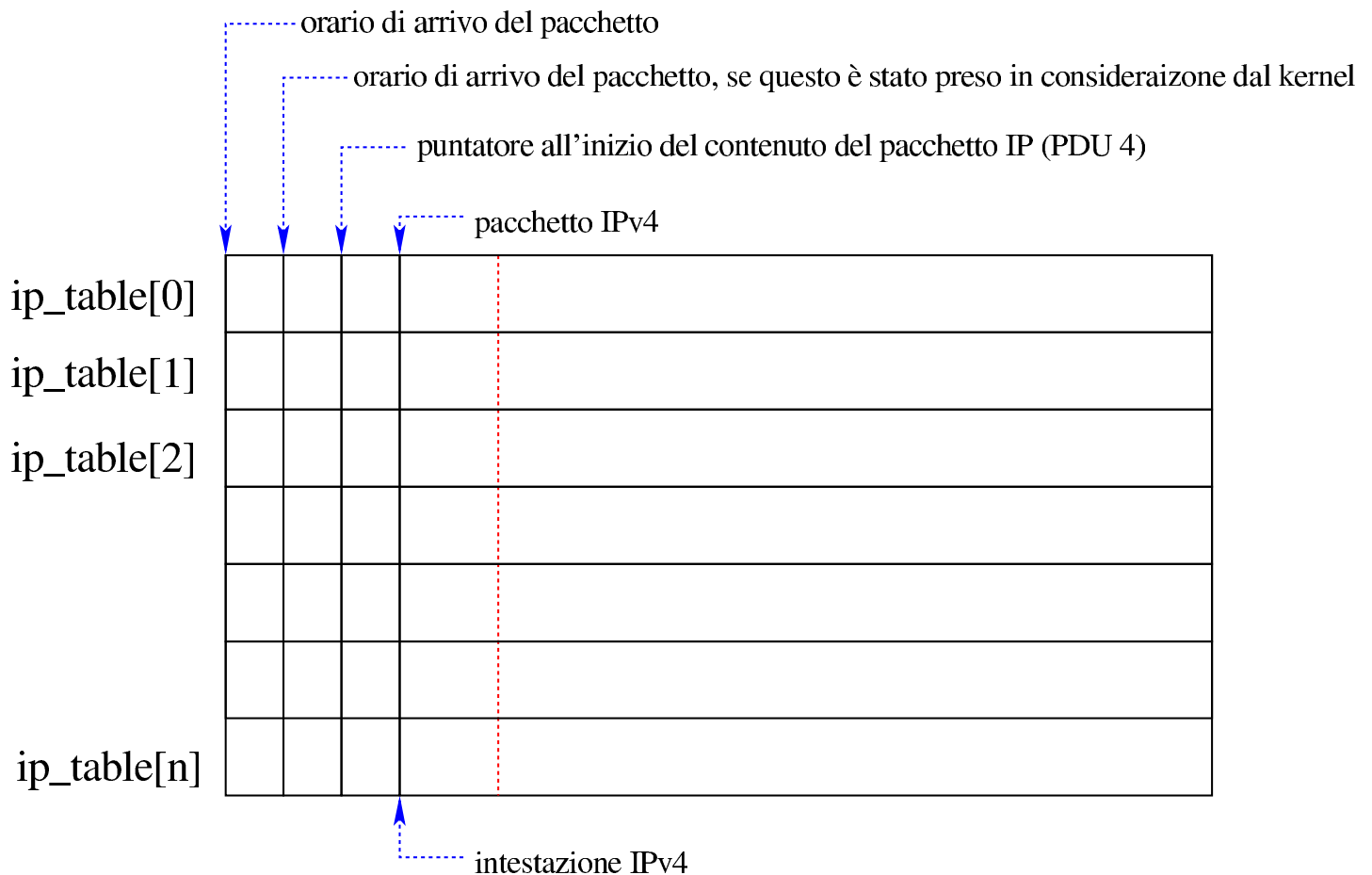
In varie situazioni si usa il tipo '**h_addr_t**', il quale rappresenta un indirizzo IPv4, a 32 bit, espresso però secondo l'architettura dell'elaboratore (*host byte order*). Questo tipo derivato si contrappone a quello standard, denominato '**in_addr_t**', il quale rappresenta lo stesso indirizzo, ma secondo l'ordinamento adatto alla trasmissione in rete (*network byte order*).

84.10.1 Tabella IPv4

«

Quando un pacchetto viene ricevuto ed è riconosciuto dalla funzione *net_rx()* come riguardante IPv4, questa chiama la funzione *ip_rx()* che lo copia nella tabella *ip_table[]*, dove rimane fino a quando viene rimpiazzato da un nuovo pacchetto, secondo il criterio per cui i pacchetti più vecchi lasciano il posto a quelli più recenti.

Figura 84.125. Struttura della tabella dei pacchetti IPv4.



Listato 84.126. Struttura di ogni elemento della tabella dei pacchetti IPv4.

```
typedef struct {
    clock_t    clock;
    clock_t    kernel_serviced;
    uint8_t    *pdu4;
    union {
        uint8_t    octet[NET_IP_MAX_PACKET_SIZE];
        struct iphdr header;
    } packet;
} ip_t;
```

Il membro *kernel_serviced* contiene inizialmente il valore zero, per poi ottenere una copia dell'orario di arrivo del pacchetto, appena questo risulta essere stato considerato dal kernel, ai fini del protocollo ICMP (in quanto il protocollo ICMP viene gestito internamente). Quando il kernel trova un pacchetto che ha l'orario di arrivo uguale a quello di elaborazione, sa così che l'ha già preso in considerazione nella propria gestione interna e non deve farci altro.

Il membro *pdu4* contiene un puntatore all'inizio del contenuto del pacchetto IPv4, ovvero a ciò che c'è nel pacchetto, dopo l'intestazione IPv4 e dopo le opzioni eventuali.

84.10.2 Ricezione di un pacchetto IPv4

La ricezione di un pacchetto IPv4 avviene per opera della funzione *ip_rx()*, la quale viene avviata da *net_rx()*, quando si accorge di avere a che fare con un pacchetto di questo tipo.

La funzione *ip_rx()* riceve due argomenti, *n* e *f*, i quali rappresentano, rispettivamente, l'indice dell'interfaccia che ha ricevuto la trama

e l'indice della trama stessa. Con questi indici, la funzione *ip_rx()* è in grado di estrapolare il pacchetto IPv4 dalla tabella *net_table[]*.

Avendo individuato l'inizio del pacchetto IPv4, verifica l'integrità del contenuto dell'intestazione con il codice di controllo relativo: se la verifica ha successo, e se non si tratta di un frammento (in quanto os32 non gestisce pacchetti frammentati), il pacchetto viene accolto e copiato nella prima posizione disponibile della tabella *ip_table[]*, annotando l'orario di arrivo.

Il pacchetto ricevuto in questo modo, dovrebbe risultare destinato a un'interfaccia del proprio sistema. Se però l'indirizzo IPv4 di destinazione non è abbinato ad alcuna interfaccia, viene trasmesso un pacchetto ICMP con il messaggio di destinazione irraggiungibile.

Se il pacchetto ricevuto risulta includere informazioni su porte UDP o TCP, viene verificato se nella tabella *sock_table[]* è prevista la ricezione nella porta che questo pacchetto dovrebbe raggiungere. Se non è così, viene trasmesso un pacchetto ICMP con il messaggio di porta non raggiungibile.

La tabella *sock_table[]* è dichiarata nel file 'kernel/fs.h' (94.5), perché le connessioni TCP e UDP, a cui si riferisce, hanno un trattamento affine a quello dei file comuni.

Alla fine, se il pacchetto risulta essere di tipo ICMP, viene avviata la funzione *icmp_rx()* perché se ne occupi; diversamente viene semplicemente copiato l'orario di ricevimento del pacchetto nel campo che rappresenta l'elaborazione dello stesso a livello IP.

84.10.3 Instradamenti



Nella directory ‘kernel/net/route/’ si trovano i file delle funzioni che consentono la gestione degli instradamenti, raccolte nel file di intestazione ‘kernel/net/route.h’ (listato 94.12.33 e successivi). Per la limitazione di os32, gli instradamenti servono in pratica solo per la trasmissione, in quanto non è previsto il funzionamento in qualità di router (quindi non si pone il problema di reindirizzare i pacchetti ricevuti).

Figura 84.127. Struttura della tabella *route_table[]* per la gestione degli instradamenti.

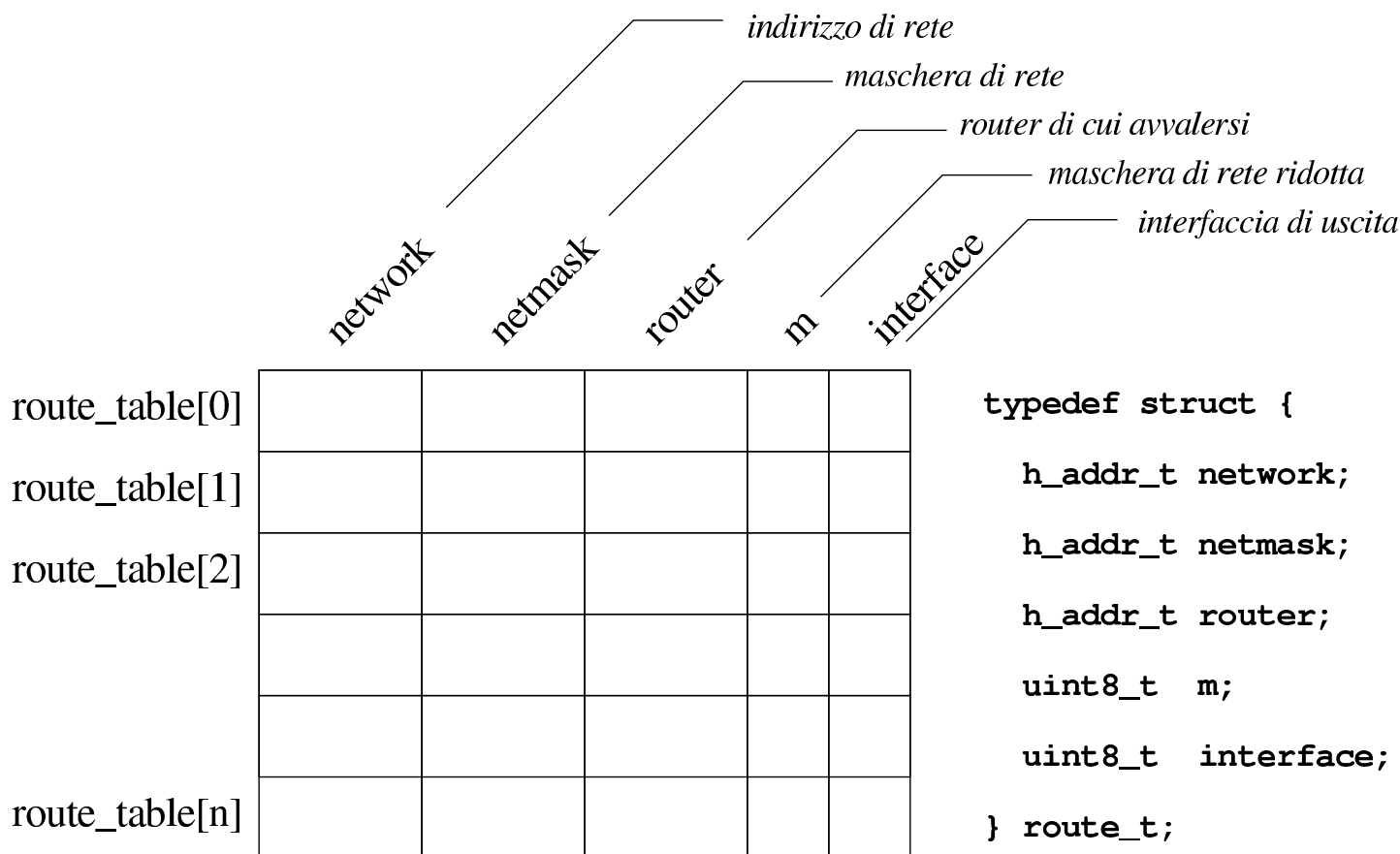


Tabella 84.128. Funzioni per la gestione della tabella degli instradamenti.

Funzione	Descrizione
<pre>void route_init (void);</pre>	<p>Inizializza la tabella <i>route_table[]</i>, predisponendo la voce <i>route_table[0]</i> per l'interfaccia locale <i>loopback</i> [94.12.34].</p>
<pre>void route_sort (void);</pre>	<p>Riordina la tabella degli instradamenti [94.12.39].</p>
<pre>h_addr_t route_remote_to_local (h_addr_t <i>remote</i>);</pre>	<p>Restituisce l'indirizzo IPv4 locale, più adatto per intrattenere una connessione con l'indirizzo remoto fornito come argomento. Questo tipo di analisi viene determinato partendo dalla tabella degli instradamenti, per determinare l'indirizzo IPv4 locale dell'interfaccia interessata dal collegamento [94.12.37].</p>

Funzione	Descrizione
<pre> h_addr_t route_remote_to_router (h_addr_t <i>remote</i>) ; </pre>	<p>Restituisce l'indirizzo IPv4 del router da utilizzare per raggiungere l'indirizzo IPv4 remoto specificato. Se l'indirizzo restituito è pari a -1 significa che non è stata ottenuta alcuna voce corrispondente, mentre se si ottiene zero significa che non c'è bisogno di router per raggiungere la destinazione [94.12.38].</p>

84.11 Gestione del protocollo ICMP

Listato [94.12.10](#) e successivi.

Quando viene ricevuto un pacchetto IPv4 che contiene un messaggio ICMP, la funzione *ip_rx()* chiama la funzione *icmp_rx()* per il trattamento di questa informazione. La funzione *icmp_rx()* verifica il tipo di messaggio e si comporta di conseguenza: a una richiesta di eco risponde con la trasmissione di un pacchetto appropriato, attraverso la funzione *icmp_tx_echo()*; a un messaggio di destinazione irraggiungibile, comunica l'informazione nella tabella *sock_table[]*, dopo aver trovato lì dentro la voce di una connessione con le caratteristiche appropriate.



Tabella 84.129. Funzioni per la gestione del protocollo ICMP.

Funzione	Descrizione
<pre>int icmp_rx (int <i>i</i>);</pre>	<p>Recepisce un pacchetto ICMP dalla tabella dei pacchetti IPv4, conoscendo l'indice di questa tabella, trattando le richieste di eco e i messaggi di destinazione irraggiungibile [94.12.11].</p>
<pre>int icmp_tx (h_addr_t <i>src</i>, h_addr_t <i>dst</i>, int <i>type</i>, int <i>code</i>, icmp_packet_t *<i>icmp</i>, size_t <i>size</i>);</pre>	<p>Completa l'intestazione di un pacchetto ICMP, con il tipo e il codice del messaggio, oltre al codice di controllo; questo pacchetto poi viene imbustato in un pacchetto IPv4 per la trasmissione [94.12.12].</p>
<pre>int icmp_tx_echo (h_addr_t <i>src</i>, h_addr_t <i>dst</i>, int <i>type</i>, int <i>code</i>, int <i>identifier</i>, int <i>sequence</i>, uint8_t *<i>data</i>, size_t <i>size</i>);</pre>	<p>Prepara un pacchetto ICMP di eco, partendo dal contenuto di questo pacchetto e dalle informazioni da aggiungere nell'intestazione ICMP; l'intestazione di questo pacchetto viene completata attraverso la funzione <i>icmp_tx()</i>, la quale poi provvede a imbustarlo in un pacchetto IPv4 per la trasmissione [94.12.13].</p>

Funzione	Descrizione
<pre data-bbox="108 363 662 838"> int icmp_tx_unreachable (h_addr_t src, h_addr_t dst, int type, int code, icmp_packet_t *icmp, size_t size); </pre>	<p data-bbox="970 159 1489 1007">Prepara un pacchetto ICMP con un messaggio di destinazione irraggiungibile, partendo dal contenuto di questo pacchetto e dalle informazioni da aggiungere nell'intestazione ICMP; l'intestazione di questo pacchetto viene completata attraverso la funzione <i>icmp_tx()</i>, la quale poi provvede a imbustarlo in un pacchetto IPv4 per la trasmissione [94.12.14].</p>

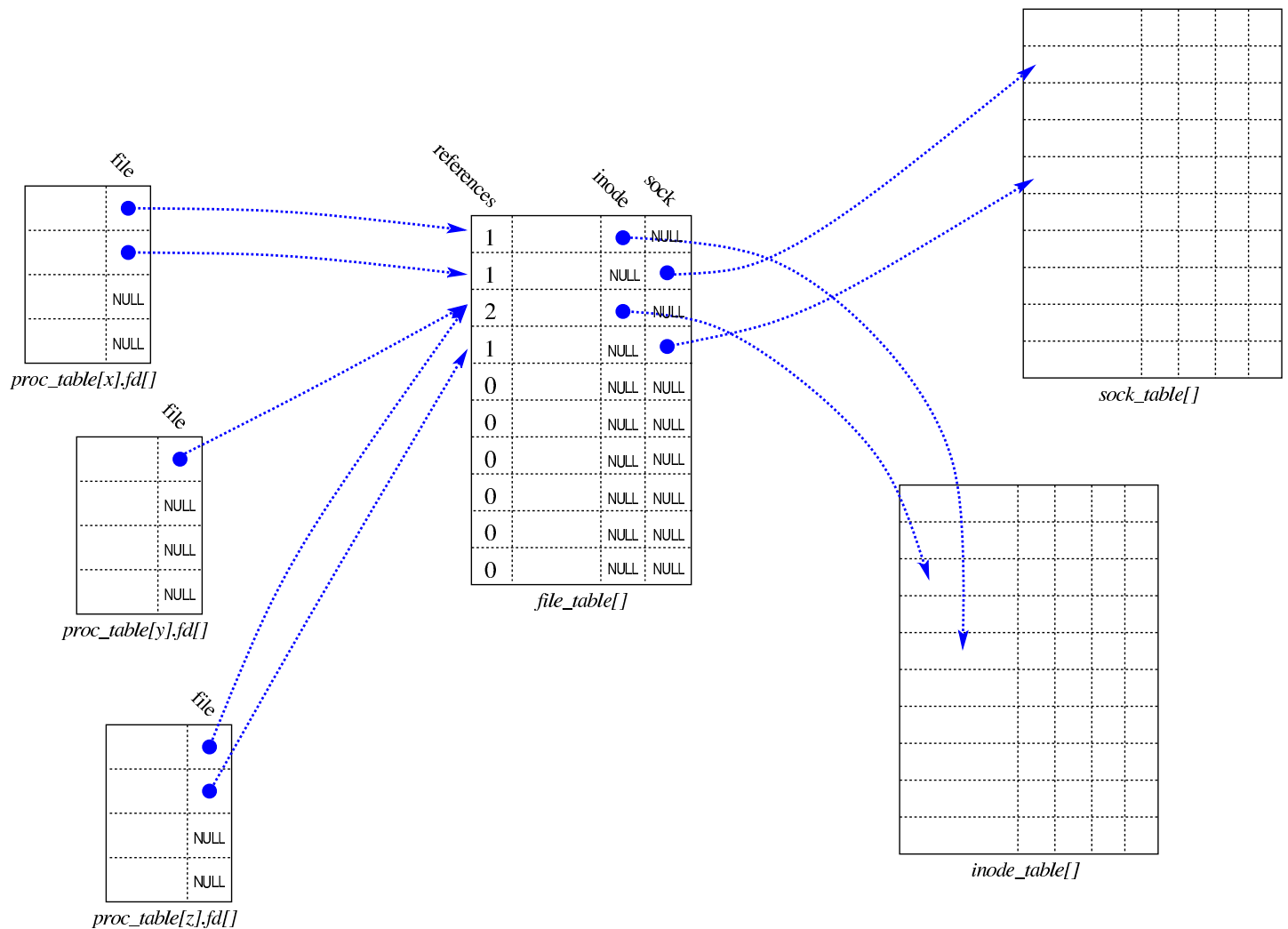
84.12 Gestione dei protocolli UDP e TCP

Listati [94.5](#), [94.12.53](#) e successivi, [94.12.40](#) e successivi.

Per la gestione dei protocolli UDP e TCP, os32 definisce una tabella in cui ogni voce descrive una connessione. Si tratta della tabella *sock_table[]* le cui voci sono collegate dalla tabella *file_table[]*. In pratica, quando si utilizza la funzione *socket()* per creare una connessione UDP o TCP, si ottiene un descrittore di file che fa riferimento a una voce nella tabella *sock_table[]*, invece che alla tabella degli inode.



Figura 84.130. Collegamento tra le tabelle dei descrittori e la tabella complessiva dei file. La tabella *proc_table[x].fd[]* rappresenta i descrittori di file del processo elaborativo *x*.



La struttura degli elementi della tabella *sock_table[]* è molto articolata. In generale si distingue una prima parte, comune ai protocolli UDP e TCP, rispetto a una seconda specifica per il protocollo TCP.

Listato 84.131. Struttura di ogni elemento della tabella dei socket.

```
struct sock
{
    int    family;
    int    type;
```

```
int          protocol;
h_addr_t    laddr;          // indirizzo locale
h_port_t    lport;         // porta locale
h_addr_t    raddr;         // indirizzo remoto
h_port_t    rport;         // porta remota
struct {
    clock_t  clock[IP_MAX_PACKETS];
} read;
uint8_t     active         : 1, // socket utilizzato?
unreach_net : 1, // rete irraggiungibile?
unreach_host: 1, // destinazione
                // irraggiungibile?
unreach_prot: 1, // protocollo
                // irraggiungibile?
unreach_port: 1; // porta irraggiungibile?

struct
{
    uint16_t  conn          : 4, // stato della connessione
    can_write : 1, // si può scrivere in
                // `send_data[]'?
    can_read  : 1, // si può leggere a
                // partire da
                // `*recv_index'?
    can_send  : 1, // si possono inviare
                // dati?
    can_recv  : 1, // si possono ricevere
                // dati?
    send_closed : 1, // il canale di
                // trasmissione
                // è chiuso?
    recv_closed : 1; // il canale di ricezione
                // è chiuso?

    //
    uint32_t  lsq[16]; // array della sequenza locale
```


Tabella 84.132. Funzioni per la gestione dei protocolli TCP e UDP.

Funzione	Descrizione
<pre>int tcp_tx_raw (h_port_t <i>sport</i>, h_port_t <i>dport</i>, uint32_t <i>seq</i>, uint32_t <i>ack_seq</i>, int <i>flags</i>, h_addr_t <i>saddr</i>, h_addr_t <i>daddr</i>, const void *<i>buffer</i>, size_t <i>size</i>);</pre>	<p>Costruisce un pacchetto TCP, utilizzando i dati forniti come argomenti della chiamata; quindi lo trasmette attraverso la funzione <i>ip_tx()</i> che a sua volta provvede a imbustarlo in un pacchetto IP prima della trasmissione effettiva. Si tratta comunque di una funzione usata soltanto per fare dei test di funzionamento [94.12.50].</p>
<pre>void tcp_show (h_addr_t <i>src</i>, h_addr_t <i>dst</i>, const struct tcphdr *<i>tcphdr</i>);</pre>	<p>Funzione diagnostica realizzata per visualizzare alcune informazioni su un pacchetto TCP, di cui si conosce il contenuto e gli indirizzi IPv4. Questa funzione viene usata prevalentemente da <i>tcp()</i>, quando si attiva la macro-variabile DEBUG [94.12.46].</p>

Funzione	Descrizione
<pre>int tcp_tx_rst (void *<i>ip_packet</i>);</pre>	Sulla base di un pacchetto IP ricevuto con un contenuto TCP, trasmette un pacchetto TCP di azzera-mento (RST) [94.12.51].
<pre>int tcp_tx_sock (void *<i>sock_item</i>);</pre>	Trasmette quanto conte-nuto nella coda del socket indicato come argomento, ammesso che il socket sia nella condizione di poter trasmettere [94.12.52].
<pre>int tcp_tx_ack (void *<i>sock_item</i>);</pre>	Trasmette un pacchetto TCP vuoto contenente la conferma di quanto rice-vuto in precedenza, sul-la base dello stato attuale del socket indicato come argomento [94.12.49].
<pre>int tcp_rx_ack (void *<i>sock_item</i>, void *<i>packet</i>);</pre>	Verifica che il pacchet-to indicato come secondo parametro, contenga una conferma valida per il soc-cket specificato come pri-mo parametro [94.12.44].

Funzione	Descrizione
<pre>int tcp_rx_data (void *<i>sock_item</i>, void *<i>packet</i>);</pre>	<p>Legge il contenuto di un pacchetto TCP e lo copia all'interno della memoria tampone del socket rappresentata da <i>sock_table[s].tcp.recv_data</i> (dove <i>s</i> è l'indice del socket considerato) [94.12.45].</p>
<pre>int tcp_connect (void *<i>sock_item</i>);</pre>	<p>Fa in modo di mettere il socket in connessione, ammesso che ciò sia possibile. Questa funzione serve a <i>s_connect()</i> [94.12.43].</p>
<pre>int tcp_close (void *<i>sock_item</i>);</pre>	<p>Fa in modo di mettere il socket nello stato di chiusura, ammesso che ciò sia possibile. Questa funzione serve a <i>s_close()</i> [94.12.42].</p>
<pre>int tcp (void);</pre>	<p>Viene chiamata da <i>proc_sch_net()</i> e si occupa di gestire lo stato di tutte le connessioni TCP in essere in quel momento [94.12.41].</p>

Funzione	Descrizione
<pre>int udp_tx (h_port_t sport, h_port_t dport, h_addr_t saddr, h_addr_t daddr, const void *buffer, size_t size);</pre>	<p>Assembla un pacchetto UDP e lo trasmette (dopo aver costruito a sua volta il pacchetto IPv4 complessivo) [94.12.54].</p>

84.12.1 UDP



Quando viene ricevuto un pacchetto IPv4 che contiene dati del protocollo UDP, dopo aver verificato che esiste effettivamente una porta UDP in attesa di ricevere nella tabella *sock_table[]*, questo viene semplicemente lasciato nella tabella *ip_table[]*, annotando soltanto che il kernel lo ha già preso in considerazione.

L'acquisizione effettiva del pacchetto UDP avviene attraverso la funzione *s_recvfrom()*, la quale costituisce la versione interna al kernel di *recvfrom()*. La funzione *s_recvfrom()*, chiamata per leggere da un socket UDP, cerca nella tabella *ip_table[]* un pacchetto corrispondente alle caratteristiche del socket, che non sia già stato preso in considerazione dal socket stesso (il membro *read.clock[i]*, dove *i* corrisponde all'indice del pacchetto trovato nella tabella *ip_table[]*, contiene l'orario di un pacchetto già letto in quella posizione: se l'orario del pacchetto contenuto nella tabella *ip_table[]* è più recente, allora deve essere letto). Se il pacchetto viene accettato, si aggiorna nel socket il valore del membro *read.clock[i]* con l'orario di ricevimento del pacchetto (per evitare di rileggerlo un'altra volta),

quindi viene copiato il contenuto del pacchetto nella destinazione specificata dagli argomenti della funzione.

Figura 84.133. Semplificazione dei punti principali del procedimento di lettura di un pacchetto UDP, attraverso la funzione *s_recvfrom()*.

```
s_recvfrom ( pid_t pid, int sfdn, void *buffer, size_t length, int flags, struct sockaddr *addrfrom, socklen_t *addrlen );
```

individua il socket
`sock = &(proc_table[pid].fd[sfdn]->file->sock)`

si suppone che si tratti di un socket UDP
`sock->type == DGRAM`
`sock->protocol == IPPROTO_UDP`

si scandisce la tabella dei pacchetti IP (con l'indice i) alla ricerca di un pacchetto UDP non ancora letto
`ip_table[i].packet.header.protocol == IPPROTO_UDP`
`ip_table[i].clock < sock->read.clock[i]`

si scandisce la tabella dei pacchetti IP alla ricerca di un pacchetto UDP non ancora letto
`udp = (struct udphdr *) &ip_table[i].packet.octet[sizeof (struct iphdr)];`
il pacchetto deve essere destinato alla stessa porta in cui in ascolto il socket, e deve essere diversa da zero
`udp->dest != 0`
`udp->dest == htons (sock->lport)`

il pacchetto deve provenire dalla porta remota che il socket ha già annotato, oppure la porta remota non deve essere ancora conosciuta
`udp->source == htons (sock->rport) || sock->rport == 0`

gli indirizzi IP devono coincidere o non devono essere conosciuti dal socket
`ip_table[i].packet.header.daddr == htonl (sock->laddr) || sock->laddr == 0`
`ip_table[i].packet.header.saddr == htonl (sock->raddr) || sock->raddr == 0`

si annota che il pacchetto è stato letto
`sock->read.clock[i] = ip_tabe[i].clock`

individua il contenuto del pacchetto UDP e lo copia dove richiesto
`data = ((uint8_t *) udp) + sizeof (struct udphdr))`

`memcpy (buffer, data, length)`

Va osservato che la lettura del pacchetto UDP, così come viene fatta da os32, si limita alla porzione specificata dalla dimensione massima della memoria tampone; se poi si esegue una nuova lettura, si cerca semplicemente un altro pacchetto, senza terminare eventualmente la lettura del precedente.

La trasmissione avviene attraverso la funzione *s_send()* (corrispondente a *send()* dal lato utente). Questa funzione, dopo aver determi-

nato che si tratta di un socket UDP, si avvale a sua volta della funzione *udp_tx()* per costruire e spedire effettivamente il pacchetto. Come nel caso della ricezione, la trasmissione riguarda un solo pacchetto, e le informazioni eccedenti sono semplicemente eliminate.

84.12.2 TCP

«

La gestione del TCP è estremamente più complessa rispetto a UDP, in quanto richiede un proprio sistema di gestione delle connessioni. La funzione *tcp()* ha il compito di scandire la tabella dei pacchetti *ip_table[]* alla ricerca di quelli che riguardano il protocollo TCP e che non sono ancora stati considerati, aggiornando lo stato delle connessioni relative. La funzione *tcp()* è chiamata a ogni interruzione da *proc_sch_net()*.

La funzione *tcp()*, quando individua nella tabella *ip_table[]* un pacchetto TCP ancora da prendere in considerazione, deve valutare le caratteristiche del pacchetto trovato in relazione allo stato della connessione eventualmente già in corso, agendo di conseguenza. Ciò significa che la funzione *tcp()* può trovarsi nella necessità di trasmettere a sua volta pacchetti TCP alla controparte, per il fine della gestione della connessione.

È importante osservare che gli indicatori *sock_table[].tcp.can_read* e *sock_table[].tcp.can_write*, necessari a controllare la lettura e la scrittura del socket con le funzioni *s_recvfrom()* e *s_send()*, sono aggiornati dalla funzione *tcp()*.

Le funzioni *s_recvfrom()* e *s_send()*, se si trovano nell'impossibilità di leggere o scrivere il socket richiesto, in condizioni normali mettono il processo relativo in pausa, in attesa di un cambiamento. Inoltre, la funzione *s_recvfrom()* deve aggiornare l'indice di lettura interno

al socket, in modo che la lettura successiva riprenda da quella posizione. In pratica, lettura e scrittura avvengono qui in modo analogo a quello di un file, in un flusso continuo di byte.

Il risveglio dei processi in attesa di leggere o scrivere un socket avviene per opera della funzione *proc_sch_net()* dopo aver avviato *tcp()* per aggiornare lo stato dei socket TCP, in base al fatto che sia stato ricevuto qualcosa o che ci sia motivo di ritenere che sia possibile scrivere attraverso un socket bloccato precedentemente in scrittura.

Esiste un grosso limite di os32, relativo alla gestione del TCP: la chiusura di una connessione elimina le informazioni relative al socket, mentre la controparte potrebbe non essere ancora pronta per recepire tale conclusione. Questa semplificazione serve a far sì che ci sia sempre corrispondenza tra il descrittore di file e il socket, mentre in un sistema reale, il socket deve poter continuare a esistere per un certo tempo, benché chiuso, anche dopo la chiusura del descrittore di file relativo.

Va poi considerato che os32 gestisce finestre TCP pari a un solo pacchetto, per cui si attende la conferma di ogni singola trasmissione dalla controparte.



85	Gestione	455
85.1	Disco fisso in un file-immagine	455
85.2	Configurazione dell'avvio e opzioni del kernel	457
85.3	Configurazione per l'uso della rete	458
85.4	Avvio di os32	460
85.5	Interazione con il kernel	461
85.6	Avvio e conclusione del sistema «normale»	465
86	Sezione 1: programmi eseguibili o comandi interni di shell 467	
86.1	os32: aaa(1)	467
86.2	os32: allocated(1)	468
86.3	os32: bbb(1)	469
86.4	os32: cat(1)	469
86.5	os32: ccc(1)	470
86.6	os32: chgrp(1)	470
86.7	os32: chmod(1)	470
86.8	os32: chown(1)	471
86.9	os32: cp(1)	472
86.10	os32: date(1)	473
86.11	os32: ed(1)	475
86.12	os32: kill(1)	476

86.13	os32: ln(1)	478
86.14	os32: login(1)	479
86.15	os32: ls(1)	480
86.16	os32: man(1)	482
86.17	os32: mkdir(1)	483
86.18	os32: mmcheck(1)	484
86.19	os32: more(1)	485
86.20	os32: nc(1)	486
86.21	os32: ps(1)	487
86.22	os32: rm(1)	489
86.23	os32: rmdir(1)	490
86.24	os32: shell(1)	491
86.25	os32: t_(1)	492
86.26	os32: touch(1)	493
86.27	os32: tty(1)	493
86.28	os32: yes(1)	494
87	Sezione 2: chiamate di sistema	497
87.1	os32: _Exit(2)	497
87.2	os32: _exit(2)	497
87.3	os32: accept(2)	498
87.4	os32: bind(2)	501
87.5	os32: brk(2)	503
87.6	os32: chdir(2)	505
87.7	os32: chmod(2)	507
87.8	os32: chown(2)	510

87.9	os32: clock(2)	512
87.10	os32: close(2)	513
87.11	os32: connect(2)	514
87.12	os32: dup(2)	516
87.13	os32: dup2(2)	517
87.14	os32: execve(2)	517
87.15	os32: fchdir(2)	520
87.16	os32: fchmod(2)	520
87.17	os32: fchown(2)	520
87.18	os32: fcntl(2)	520
87.19	os32: fork(2)	524
87.20	os32: fstat(2)	525
87.21	os32: getcwd(2)	526
87.22	os32: getgid(2)	527
87.23	os32: geteuid(2)	528
87.24	os32: getpgrp(2)	528
87.25	os32: getpid(2)	529
87.26	os32: getppid(2)	530
87.27	os32: getuid(2)	530
87.28	os32: ipconfig(2)	531
87.29	os32: kill(2)	533
87.30	os32: link(2)	534
87.31	os32: listen(2)	537
87.32	os32: longjmp(2)	538
87.33	os32: lseek(2)	539

87.34	os32: mkdir(2)	540
87.35	os32: mknod(2)	543
87.36	os32: mount(2)	545
87.37	os32: open(2)	547
87.38	os32: pipe(2)	552
87.39	os32: read(2)	553
87.40	os32: recvfrom(2)	555
87.41	os32: rmdir(2)	557
87.42	os32: routeadd(2)	559
87.43	os32: routedel(2)	560
87.44	os32: sbrk(2)	561
87.45	os32: send(2)	562
87.46	os32: setegid(2)	564
87.47	os32: seteuid(2)	564
87.48	os32: setgid(2)	564
87.49	os32: setjmp(2)	567
87.50	os32: setpgrp(2)	570
87.51	os32: setuid(2)	572
87.52	os32: signal(2)	574
87.53	os32: sleep(2)	576
87.54	os32: socket(2)	577
87.55	os32: stat(2)	580
87.56	os32: sys(2)	586
87.57	os32: stime(2)	588
87.58	os32: tcgetattr(2)	588

87.59	os32: time(2)	593
87.60	os32: umask(2)	594
87.61	os32: umount(2)	596
87.62	os32: unlink(2)	596
87.63	os32: wait(2)	597
87.64	os32: write(2)	598
87.65	os32: z(2)	600
87.66	os32: z_perror(2)	602
87.67	os32: z_printf(2)	602
87.68	os32: z_vprintf(2)	602
88	Sezione 3: funzioni di libreria	603
88.1	os32: _gcc(3)	603
88.2	os32: abort(3)	603
88.3	os32: abs(3)	604
88.4	os32: access(3)	605
88.5	os32: asctime(3)	607
88.6	os32: assert(3)	607
88.7	os32: atexit(3)	608
88.8	os32: atoi(3)	610
88.9	os32: atol(3)	611
88.10	os32: basename(3)	611
88.11	os32: byteorder(3)	613
88.12	os32: clearerr(3)	614
88.13	os32: closedir(3)	615
88.14	os32: creat(3)	616

88.15	os32: ctime(3)	617
88.16	os32: dirname(3)	620
88.17	os32: div(3)	620
88.18	os32: endgrent(3)	622
88.19	os32: endpwent(3)	622
88.20	os32: errno(3)	622
88.21	os32: exec(3)	634
88.22	os32: execl(3)	637
88.23	os32: execl(3)	637
88.24	os32: execlp(3)	637
88.25	os32: execv(3)	637
88.26	os32: execvp(3)	637
88.27	os32: exit(3)	637
88.28	os32: fclose(3)	637
88.29	os32: feof(3)	638
88.30	os32: ferror(3)	639
88.31	os32: fflush(3)	640
88.32	os32: fgetc(3)	641
88.33	os32: fgetpos(3)	643
88.34	os32: fgets(3)	644
88.35	os32: fileno(3)	646
88.36	os32: fopen(3)	647
88.37	os32: fprintf(3)	650
88.38	os32: fputc(3)	651
88.39	os32: fputs(3)	652

88.40	os32: fread(3)	654
88.41	os32: free(3)	654
88.42	os32: freopen(3)	655
88.43	os32: fscanf(3)	655
88.44	os32: fseek(3)	655
88.45	os32: fseeko(3)	657
88.46	os32: fsetpos(3)	657
88.47	os32: ftell(3)	657
88.48	os32: ftello(3)	658
88.49	os32: fwrite(3)	659
88.50	os32: getc(3)	659
88.51	os32: getchar(3)	660
88.52	os32: getenv(3)	660
88.53	os32: getgrent(3)	661
88.54	os32: getgrnam(3)	664
88.55	os32: getpwuid(3)	666
88.56	os32: getopt(3)	666
88.57	os32: getpwent(3)	672
88.58	os32: getpwnam(3)	675
88.59	os32: getpwuid(3)	678
88.60	os32: gets(3)	678
88.61	os32: gmtime(3)	678
88.62	os32: htonl(3)	678
88.63	os32: htons(3)	678
88.64	os32: imaxabs(3)	678

88.65	os32: imaxdiv(3)	678
88.66	os32: inet_ntop(3)	678
88.67	os32: inet_pton(3)	680
88.68	os32: input_line(3)	681
88.69	os32: isatty(3)	683
88.70	os32: labs(3)	684
88.71	os32: ldiv(3)	684
88.72	os32: llabs(3)	684
88.73	os32: lldiv(3)	684
88.74	os32: major(3)	684
88.75	os32: makedev(3)	684
88.76	os32: malloc(3)	685
88.77	os32: memccpy(3)	687
88.78	os32: memchr(3)	688
88.79	os32: memcmp(3)	689
88.80	os32: memcpy(3)	690
88.81	os32: memmove(3)	691
88.82	os32: memset(3)	691
88.83	os32: minor(3)	692
88.84	os32: mktime(3)	692
88.85	os32: namep(3)	693
88.86	os32: ntohl(3)	694
88.87	os32: ntohs(3)	694
88.88	os32: offsetof(3)	694
88.89	os32: opendir(3)	695

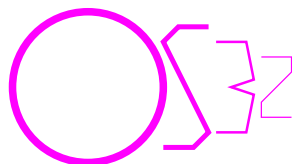
88.90	os32: perror(3)	697
88.91	os32: printf(3)	698
88.92	os32: putc(3)	705
88.93	os32: putchar(3)	705
88.94	os32: putenv(3)	705
88.95	os32: puts(3)	707
88.96	os32: qsort(3)	707
88.97	os32: rand(3)	709
88.98	os32: readdir(3)	710
88.99	os32: realloc(3)	711
88.100	os32: rewind(3)	711
88.101	os32: rewinddir(3)	712
88.102	os32: scanf(3)	713
88.103	os32: setbuf(3)	722
88.104	os32: setenv(3)	723
88.105	os32: setgrent(3)	725
88.106	os32: setpwent(3)	725
88.107	os32: setvbuf(3)	725
88.108	os32: snprintf(3)	725
88.109	os32: sprintf(3)	725
88.110	os32: srand(3)	726
88.111	os32: sscanf(3)	726
88.112	os32: stdio(3)	726
88.113	os32: strcat(3)	728
88.114	os32: strchr(3)	730

88.115	os32: strcmp(3)	731
88.116	os32: strcoll(3)	732
88.117	os32: strcpy(3)	732
88.118	os32: strcspn(3)	733
88.119	os32: strdup(3)	733
88.120	os32: strerror(3)	734
88.121	os32: strlen(3)	735
88.122	os32: strncat(3)	736
88.123	os32: strncmp(3)	736
88.124	os32: strncpy(3)	736
88.125	os32: strpbrk(3)	736
88.126	os32: strrchr(3)	737
88.127	os32: strspn(3)	738
88.128	os32: strstr(3)	739
88.129	os32: strtok(3)	739
88.130	os32: strtol(3)	743
88.131	os32: strtoul(3)	745
88.132	os32: strxfrm(3)	746
88.133	os32: ttyname(3)	747
88.134	os32: unsetenv(3)	748
88.135	os32: vfprintf(3)	748
88.136	os32: vscanf(3)	748
88.137	os32: vprintf(3)	748
88.138	os32: vscanf(3)	751
88.139	os32: vsnprintf(3)	753

88.140	os32: vsprintf(3)	754
88.141	os32: vsscanf(3)	754
89	Sezione 4: file speciali	755
89.1	os32: console(4)	755
89.2	os32: ata(4)	756
89.3	os32: kmem_arp(4)	757
89.4	os32: kmem_file(4)	758
89.5	os32: kmem_inode(4)	759
89.6	os32: kmem_mmp(4)	759
89.7	os32: kmem_net(4)	760
89.8	os32: kmem_ps(4)	761
89.9	os32: kmem_route(4)	762
89.10	os32: kmem_sb(4)	763
89.11	os32: mem(4)	763
89.12	os32: null(4)	764
89.13	os32: port(4)	765
89.14	os32: tty(4)	766
89.15	os32: zero(4)	766
90	Sezione 5: formato dei file e convenzioni	769
90.1	os32: group(5)	769
90.2	os32: inittab(5)	770
90.3	os32: issue(5)	771
90.4	os32: passwd(5)	771
91	Sezione 7: varie	773

91.1	os32: environ(7)	773
91.2	os32: socket(7)	775
91.3	os32: undocumented(7)	776
92	Sezione 8: comandi per l'amministrazione del sistema	777
92.1	os32: arp(8)	777
92.2	os32: getty(8)	778
92.3	os32: http(8)	779
92.4	os32: init(8)	780
92.5	os32: ipconfig(8)	781
92.6	os32: MAKEDEV(8)	782
92.7	os32: mount(8)	783
92.8	os32: ping(8)	785
92.9	os32: route(8)	786
92.10	os32: umount(8)	787
93	Sezione 9: kernel	789
93.1	os32: arp(9)	795
93.2	os32: ata(9)	797
93.3	os32: blk(9)	802
93.4	os32: dev(9)	808
93.5	os32: dm(9)	823
93.6	os32: fs(9)	823
93.7	os32: ibm_i386(9)	891
93.8	os32: icmp(9)	897
93.9	os32: ip(9)	897

93.10	os32: kbd(9)	899
93.11	os32: lib_k(9)	900
93.12	os32: lib_s(9)	901
93.13	os32: main(9)	902
93.14	os32: memory(9)	902
93.15	os32: multiboot(9)	905
93.16	os32: ne2k(9)	907
93.17	os32: net(9)	909
93.18	os32: part(9)	912
93.19	os32: pci(9)	912
93.20	os32: proc(9)	913
93.21	os32: route(9)	950
93.22	os32: screen(9)	952
93.23	os32: tcp(9)	955
93.24	os32: tty(9)	959



Gestione

os32, come già os16, ha due modalità di funzionamento: si può interagire direttamente con il kernel, oppure si può avviare il processo `'init'` e procedere con l'organizzazione consueta di un sistema Unix tradizionale. La modalità di colloquio diretto con il kernel è servita per consentire lo sviluppo di os32 e potrebbe essere utile per motivi di studio. Va osservato che durante l'interazione diretta con il kernel si dispone di una sola console, mentre quando si avvia `'init'` si possono avere delle console virtuali in base alla configurazione.

Video <http://www.youtube.com/watch?v=Bc33a2-NRzM>

85.1 Disco fisso in un file-immagine

os32 è distribuito in modo da funzionare con l'ausilio di Qemu o Bochs, installato in un disco fisso virtuale, rappresentato da un file-immagine. Tale file-immagine, denominato `'disk.hda'` è suddiviso in almeno due partizioni: la prima è necessaria per l'avvio e il suo formato dipende dal sistema di avvio installato (dovrebbe essere SYSLINUX, pertanto il file system dovrebbe essere di tipo Dos-FAT); la seconda è di tipo Minix-1 e ospita il sistema os32.

La creazione e l'accesso alle partizioni contenute nel file `'disk.hda'` è un po' complicato da un sistema GNU/Linux, perché occorre estrapolarle di volta in volta in file indipendenti, per aggregarle poi nuovamente in un file-immagine unico. Per queste operazioni si usano degli script già predisposti:

Script	Descrizione
<code>file_image_functions</code>	Contiene delle funzioni utilizzate dagli altri script [94.1.5].

Script	Descrizione
<code>fdisk file</code>	Consente di utilizzare il programma 'fdisk' sul file-immagine indicato, con l'ausilio delle funzioni contenute nel file 'file_image_functions' [94.1.4].
<code>format file n dos minix</code>	Consente di inizializzare la partizione <i>n</i> -esima del file-immagine indicato come primo argomento, con un file system di tipo Dos-FAT o di tipo Minix-1 [94.1.6].
<code>syslinux file n</code>	Installa SYSLINUX nella partizione <i>n</i> del file-immagine indicato [94.1.10].
<code>mount</code> <code>umount</code>	Innesta o stacca le partizioni contenute nel file-immagine 'disk.hda' utilizzando le directory '/mnt/disk.hda.<i>n</i>' che però devono già essere disponibili.

Quando si compila il sistema con lo script ***makeit.mer*** o con ***makeit.sep***, al termine del processo questo tenta di copiare il kernel **'kimage'** nella directory **'/mnt/disk.hda1/'** e poi il sistema nella directory **'/mnt/disk.hda2/'**. Queste due directory dovrebbero innestare rispettivamente le prime due partizioni del file-immagine **'disk.hda'**; tuttavia, occorre disporre dei privilegi necessari, come dire che occorre fare questo in qualità di utente **'root'**.

La copia del sistema nella seconda partizione non provvede però a produrre i file di dispositivo necessari nella directory **'dev/'**. Eventualmente, all'interno di tale directory si ottiene lo script **'*.makedev*'**, da avviare attraverso il sistema GNU/Linux.

Va comunque ricordato che prima di avviare os32 è necessario distaccare gli innesti delle partizioni del file-immagine, perché os32 funziona utilizzando il file-immagine nel suo complesso, mentre

l'innesto delle partizioni comporta l'estrapolazione delle partizioni e la successiva riaggregazione; inoltre, os32 non ha un alcun sistema di arresto del sistema, per cui è facile rendere incoerente il file system, quindi è bene fare spesso delle copie del file-immagine ed essere comunque pronti a ricostruirlo.

85.2 Configurazione dell'avvio e opzioni del kernel

Si utilizza SYSLINUX nella prima partizione per l'avvio di os32. Premesso che il kernel di os32 è contenuto nel file 'kimage' e si colloca assieme ai file di SYSLINUX, nella configurazione contenuta nel file 'syslinux.cfg' (relativo a SYSLINUX stesso) si leggono le istruzioni seguenti:

```
TIMEOUT 10
PROMPT 1
DEFAULT os32
#
LABEL os32
  KERNEL mboot.c32
  APPEND kimage net1=1,172.21.11.16,16 ↵
↵      route0=0.0.0.0,0,172.21.11.18,1 ↵
↵      route1=172.21.254.254,32,172.21.11.18,1
```

Con le opzioni passate al kernel di os32 si ottiene la configurazione dell'interfaccia di rete '**net1**' ('**net0**' corrisponde all'interfaccia *loopback*), con l'indirizzo 172.21.11.16, maschera di rete 255.255.0.0 e gli instradamenti necessari.

Opzione del kernel	Descrizione
<code>net<i>i</i>=<i>i</i>, ipv4, <i>m</i></code>	Dichiara l'interfaccia di rete ' net<i>i</i> ', corrispondente all'indice <i>i</i> nella tabella delle interfacce, con l'indirizzo IPv4 specificato da ipv4 e la maschera di rete lunga <i>m</i> bit.

Opzione del kernel	Descrizione
<code>route <i>n=dest, m, router, i</i></code>	Dichiara la destinazione rappresentata dall'indirizzo <i>dest</i> e la maschera di rete composta da <i>m</i> bit, raggiungibile attraverso l'indirizzo <i>router</i> corrispondente all'interfaccia <i>i</i> nella tabella delle interfacce.

Volendo tradurre le opzioni che appaiono nell'esempio in comandi di un sistema GNU/Linux, sarebbe come se fosse stato scritto:

```
# ifconfig eth1 172.21.11.16 netmask 255.255.0.0 [Invio]

# route add -net 0.0.0.0 netmask 0.0.0.0 gw 172.21.11.18 ↵
↵      dev eth1 [Invio]

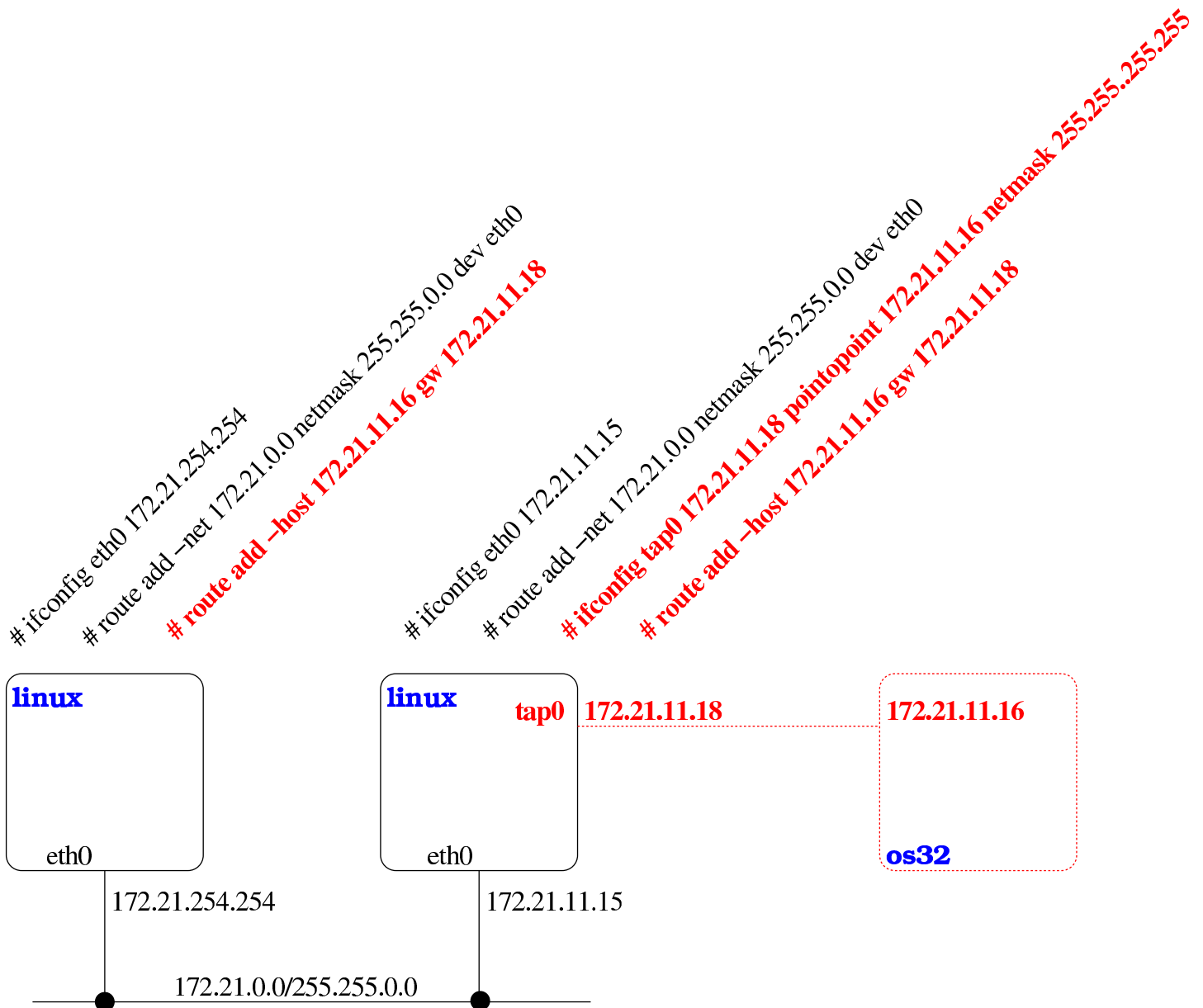
# route add -host 172.21.254.254 gw 172.21.11.18 ↵
↵      dev eth1 [Invio]
```

85.3 Configurazione per l'uso della rete

«

Per poter utilizzare le funzionalità di rete di os32, attraverso l'emulatore, è necessario predisporre nel sistema ospitante un'interfaccia di rete virtuale, connessa con quella di os32 attraverso una connessione punto-punto (sempre virtuale). Gli script che accompagnano os32 prevedono l'uso con Qemu o Bochs, in un sistema GNU/Linux, creando una connessione come quella schematizzata nella figura successiva.

Figura 85.4. Ipotesi di collegamento di os32 con il sistema ospitante e con il resto della rete locale: l'elaboratore che esegue os32 all'interno di un emulatore è quello corrispondente all'indirizzo 172.21.11.15. Le istruzioni che riguardano la configurazione necessaria a connettersi con os32 appaiono in rosso.



Il sistema os32, a sua volta, si configura esclusivamente attraverso le opzioni di avvio, come descritto nella sezione precedente.

85.4 Avvio di os32



L'avvio di os32 è un po' complicato, a causa della necessità di predisporre la rete nel modo appropriato, nel sistema ospitante, ma in pratica si usa lo script 'qemu' o lo script 'bochs', i quali si avvalgono a loro volta di 'tap0', per creare l'interfaccia di rete virtuale 'tap0' nel sistema ospitante. Tuttavia, l'avvio deve avvenire con i privilegi dell'utente 'root' per poter predisporre le funzionalità di rete; pertanto va usato un programma come 'gksu' per ottenere tali privilegi:

```
$ gksu ./qemu [Invio]
```

Oppure:

```
$ gksu ./bochs [Invio]
```

Negli esempi gli script vengono avviati indicando il percorso, per evitare che vengano confusi con i nomi dei file eseguibili di Qemu e di Bochs, i quali si trovano presumibilmente nella directory '/usr/bin/'.

Script	Descrizione
tap0	Predisporre l'interfaccia 'tap0' con l'instadamento necessario a raggiungere os32; questo script viene utilizzato da Bochs o da Qemu [94.1.11].

Script	Descrizione
qemu	Avvia Qemu con le opzioni appropriate per eseguire os32 nel file-immagine 'disk.hda', configurando la rete in modo appropriato. Tra le opzioni dell'avvio di Qemu si trova la richiesta di utilizzare lo script 'tap0' per la configurazione della rete nel sistema ospitante [94.1.9].
bochs	Avvia Bochs con le opzioni appropriate per eseguire os32 nel file-immagine 'disk.hda', configurando la rete in modo appropriato. Tra le opzioni dell'avvio di Bochs si trova la richiesta di utilizzare lo script 'tap0' per la configurazione della rete nel sistema ospitante [94.1.2].

85.5 Interazione con il kernel

L'avvio di os32 passa, in ogni caso, per una prima fase di colloquio con il kernel. Si ottiene un menù e si possono premere semplicemente dei tasti, seguiti però da [*Invio*], secondo l'elenco previsto, per ottenere delle azioni molto semplici. In questa fase il disco da cui risulta avviato il kernel è già innestato ed è prevista la possibilità di avviare tre programmi: '/bin/aaa', '/bin/bbb' e '/bin/ccc'. In tal modo, si ha la possibilità di avviare qualcosa, a titolo diagnostico, prima dello stesso '**init**' ('/bin/init').



Figura 85.6. Aspetto di os32 in funzione, con il menù in evidenza.

```

os32 build 20AAMMGHm ram 130048 Kibyte
[ata_init] ATA drive 0 size 8064 Kib
[ata_drq] ERROR: drive 2 error
[dm_init] ATA drive=0 total sectors=16128
[dm_init] partition type=0c start sector=63 total sectors=2961
[dm_init] partition type=81 start sector=3024 total sectors=13104
.------.
| h      show this menu                               .------.|
| t      show internal timer values                  | all commands ||
| f      fork the kernel                             | followed by  ||
| m      memory map (HEX)                            | [Enter]      ||
| g|G    show GDT table first 21+21 items            `------'|
| i|I    show IDT table first 21+21 items            |               |
| p      process status list                          |               |
| s      super block list                             |               |
| n      list of active inodes                        |               |
| 1..9   kill process  1 to 9                       |               |
| A..F   kill process 10 to 15                       |               |
| a..c   run programs `/bin/aaa' to `/bin/ccc' in parallel |
| x      exit interaction with kernel and start `/bin/init' |
| q      quit kernel                                  |               |
`------'

```

Le funzioni principali disponibili in questa modalità diagnostica sono riassunte nella tabella successiva. È importante tenere presente che, a differenza di os16, anche in questa fase i comandi sono conclusi con la pressione del tasto [*Invio*].

Comando	Risultato
h	Mostra il menù di funzioni disponibili.
t	Mostra i valori gestiti internamente dell'orologio del kernel.

Comando	Risultato
f	Esegue una biforcazione del kernel, nella quale, il processo figlio si limita a mostrare ripetutamente il proprio numero di processo.
m	Mostra la mappa della memoria, elencando le aree continue utilizzate.
g G	Mostra le prime voci della tabella GDT, in binario.
i I	Mostra le prime voci della tabella IDT, in binario.
p	Mostra la situazione dei processi e altre informazioni.
s	Mostra delle informazioni sul super blocco.
n	Mostra l'elenco degli inode attivi.
1	Invia il segnale ' SIGKILL ' al processo numero uno.
2 3 4 5 6 7 8 9 A B C D E F	Invia il segnale ' SIGTERM ' al processo con il numero corrispondente, da 2 a 15.
a b c	Avvia il programma '/bin/aaa', '/bin/bbb' o '/bin/ccc'.
x	Termina il ciclo e successivamente si passa all'avvio di '/bin/init'.
q	Ferma il sistema.

Figura 85.8. Aspetto di os32 in funzione mentre visualizza la tabella dei processi avviati e la mappa della memoria.

```

c
abaabaaba
p
pp  p pg          T * 0x1000 D * 0x1000 stack
id id rp  tty  uid  euid  suid  usage  s  addr  size  addr  size  pointer  name
0  0  0  0000    0    0    0  00.03  R  00000  028e  00000  0000  028eb2c  os32 kernel
0  1  0  0000    0    0    0  00.09  r  0051e  000e  0052c  002d  002cf88  /bin/ccc
1  2  0  0000   10   10   10  00.00  s  002bc  000e  002ca  002d  002cf34  /bin/aaa
1  3  0  0000   11   11   11  00.00  s  002f7  000e  00305  002d  002cf34  /bin/bbb
ab
m
Hex mem map, blocks of 1000: 0-28f 2bc-332 51e-559
aabaab_

```

Con il comando **'x'** il ciclo termina e il kernel avvia **'/bin/init'**, ma prima di farlo occorre che non ci siano altri processi in funzione, perché **'init'** deve assumere il ruolo di processo 1, ovvero il primo dopo il kernel.

Figura 85.9. Aspetto di os32 in funzione con il menù in evidenza, dopo aver dato il comando **'x'** per avviare **'init'**.

```
os32 build 20AAMMGHHmm ram 130048 Kibyte
[ata_init] ATA drive 0 size 16 Kib
[ata_init] ATA drive 1 size 16 Kib
.------.
| h      show this menu                               |-----|.
| t      show internal timer values                   | all commands ||
| f      fork the kernel                             | followed by  ||
| m      memory map (HEX)                            | [Enter]      ||
| p      process status list                         |'-----'|
| s      super block list                             |
| n      list of active inodes                       |
| 1..9   kill process  1 to 9                       |
| A..F   kill process 10 to 15                      |
| a..c   run programs '/bin/aaa' to '/bin/ccc' in parallel |
| x      exit interaction with kernel and start '/bin/init' |
| q      quit kernel                                  |
|-----|
x
init
os32: a basic os. [Ctrl q], [Ctrl r], [Ctrl s], [Ctrl t] to change
console.
This is terminal /dev/console0
Log in as "root" or "user" with password "ciao" :-)
login:
```

85.6 Avvio e conclusione del sistema «normale»

Se non si intende operare direttamente con il kernel, come descritto nella sezione precedente, con il comando **'x'** si avvia **'init'**.

Il programma **'init'** legge il file **'/etc/inittab'** e sulla base del suo contenuto, avvia uno o più processi **'getty'**, per la gestione dei vari terminali disponibili (si tratta comunque soltanto di console virtuali).

Il programma **'getty'** apre il terminale che gli viene indicato come opzione della chiamata (da **'init'** che lo determina in base al contenuto di **'/etc/inittab'**), facendo in modo che sia associato al descrittore zero (standard input). Quindi, dopo aver visualizzato il contenuto del file **'/etc/issue'**, mostra un proprio messaggio e avvia il programma **'login'**.

Il programma **'login'** prende il posto di **'getty'** che così scompare dall'elenco dei processi. **'login'** procede chiedendo all'utente di identificarsi, utilizzando il file **'/etc/passwd'** per verificare le credenziali di accesso. Se l'identificazione ha successo, viene avviata la shell definita nel file **'/etc/passwd'** per l'utente, in modo da sostituirsi al programma **'login'**, il quale scompare a sua volta dall'elenco dei processi.

Attraverso la shell è possibile interagire con il sistema operativo, secondo la modalità «normale», nei limiti delle possibilità di os32. Quando la shell termina di funzionare, **'init'** riavvia **'getty'**.

Per cambiare console virtuale si possono usare le combinazioni **[Ctrl q]**, **[Ctrl r]**, **[Ctrl s]** e **[Ctrl t]**, ma bisogna considerare che dipende dalla configurazione del file **'/etc/inittab'** se effettivamente vengono attivate tutte queste console.

Per concludere l'attività del sistema, basta concludere il funzionamento delle varie sessioni di lavoro (la shell finisce di funzionare con il comando interno **'exit'**) e non c'è bisogno di altro; pertanto, non è previsto l'uso di comandi come **'halt'** o **'shutdown'** e, d'altro canto, le operazioni di scrittura nel file system sono sincrone, in modo tale da non richiedere accorgimenti particolari per la chiusura delle attività.

Sezione 1: programmi eseguibili o comandi interni di shell



86.1 os32: aaa(1)

NOME

‘**aaa**’, ‘**bbb**’, ‘**ccc**’ - programmi elementari avviabili direttamente dal kernel

SINTASSI

```
aaa
```

```
bbb
```

```
ccc
```

DESCRIZIONE

‘**aaa**’ e ‘**bbb**’ si limitano a visualizzare una lettera, rispettivamente «a» e «b», attraverso lo standard output, a intervalli regolari. Precisamente, ‘**aaa**’ lo fa a intervalli di un secondo, mentre ‘**bbb**’ a intervalli di due secondi. Il lavoro di ‘**aaa**’ e di ‘**bbb**’ si conclude dopo l’emissione, rispettivamente, di 60 e 30 caratteri, pertanto nel giro di un minuto di tempo si esaurisce il loro compito.

Il programma ‘**ccc**’ è diverso, ma nasce per lo stesso scopo: controllare la gestione dei processi di os32. Questo programma si



limita ad avviare, ‘**aaa**’ e ‘**bbb**’, come propri processi-figli, rimanendo in funzione, senza fare nulla. Pertanto, se si usa ‘**ccc**’, il suo processo deve essere eliminato in modo esplicito, perché da solo non si concluderebbe mai.

Questi programmi sono indicati soprattutto per l’uso di os32 nella modalità interattiva che precede il funzionamento normale del sistema operativo, per la verifica della gestione dei processi.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [96.1.12]

‘`applic/crt0.sep.s`’ [96.1.13]

‘`applic/aaa.c`’ [96.1.2]

‘`applic/bbb.c`’ [96.1.5]

‘`applic/ccc.c`’ [96.1.7]

86.2 os32: `allocated(1)`

«

NOME

‘**allocated**’ - mappa della memoria allocata

SINTASSI

```
allocated
```

DESCRIZIONE

Il programma ‘**allocated**’ si limita a leggere dal file di dispositivo ‘`/dev/kmem_map`’, producendo un elenco delle zone di memoria continue già in uso.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [96.1.12]

‘`applic/crt0.sep.s`’ [96.1.13]

‘`applic/allocated.c`’ [96.1.3]

VEDERE ANCHE

mmcheck(1) [86.18].

86.3 os32: bbb(1)

Vedere *aaa(1)* [86.1].

86.4 os32: cat(1)

NOME

‘**cat**’ - emissione del contenuto di uno o più file attraverso lo standard output

SINTASSI

```
cat [file] ...
```

DESCRIZIONE

‘**cat**’ legge il contenuto dei file indicati come argomento e li emette attraverso lo standard output, concatenati assieme in un unico flusso.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [96.1.12]

‘`applic/crt0.sep.s`’ [96.1.13]

‘`applic/cat.c`’ [96.1.6]

VEDERE ANCHE

more(1) [[86.19](#)], *ed(1)* [[86.11](#)].

86.5 os32: ccc(1)

« Vedere *aaa(1)* [[86.1](#)].

86.6 os32: chgrp(1)

«

NOME

‘**chgrp**’ - cambiamento del gruppo proprietario di un file

SINTASSI

```
chgrp nome_gruppo file...
```

```
chgrp gid file...
```

DESCRIZIONE

‘**chgrp**’ cambia l’utente proprietario dei file indicati. Il nuovo proprietario da attribuire può essere indicato per nome o per numero.

FILE SORGENTI

‘*applic/crt0.mer.s*’ [[96.1.12](#)]

‘*applic/crt0.sep.s*’ [[96.1.13](#)]

‘*applic/chgrp.c*’ [[96.1.8](#)]

VEDERE ANCHE

chgrp(1) [[86.8](#)].

chmod(1) [[86.7](#)].

86.7 os32: chmod(1)



NOME

‘**chmod**’ - cambiamento della modalità dei permessi dei file

SINTASSI

```
chmod mod_ottale file...
```

DESCRIZIONE

‘**chmod**’ cambia la modalità dei permessi associati ai file indicati, in base al numero ottale indicato come primo argomento.

NOTE

Questa versione di ‘**chmod**’ non permette di indicare la modalità dei permessi in forma simbolica.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [[96.1.12](#)]

‘`applic/crt0.sep.s`’ [[96.1.13](#)]

‘`applic/chmod.c`’ [[96.1.9](#)]

VEDERE ANCHE

chown(1) [[86.8](#)].

86.8 os32: chown(1)



NOME

‘**chown**’ - cambiamento del proprietario di un file

SINTASSI

```
chown nome_utente file...
```

```
chown uid file...
```

DESCRIZIONE

‘**chown**’ cambia l’utente proprietario dei file indicati. Il nuovo proprietario da attribuire può essere indicato per nome o per numero.

FILE SORGENTI

‘*applic/crt0.mer.s*’ [[96.1.12](#)]

‘*applic/crt0.sep.s*’ [[96.1.13](#)]

‘*applic/chown.c*’ [[96.1.10](#)]

VEDERE ANCHE

chgrp(1) [[86.6](#)].

chmod(1) [[86.7](#)].

86.9 os32: cp(1)

«

NOME

‘**cp**’ - copia dei file

SINTASSI

```
cp file_orig file_nuovo...
```



```
cp file... directory_dest...
```

DESCRIZIONE

‘**cp**’ copia uno o più file. Se l’ultimo argomento è costituito da una directory esistente, la copia produce dei file con lo stesso nome degli originali, all’interno della directory; se l’ultimo argomento non è una directory già esistente, ci può essere un solo file da copiare, intendendo che si voglia creare una copia con quel nome specificato.

DIFETTI

Non è possibile copiare oggetti diversi dai file puri e semplici; quindi, la copia ricorsiva di una directory non è ammissibile.

FILE SORGENTI

‘*applic/crt0.mer.s*’ [96.1.12]

‘*applic/crt0.sep.s*’ [96.1.13]

‘*applic/cp.c*’ [96.1.11]

VEDERE ANCHE

touch(1) [86.26], *mkdir(1)* [86.17].

86.10 os32: date(1)

NOME

‘**date**’ - visualizzazione o impostazione della data e dell’ora di sistema

SINTASSI

```
date [MMGGhhmm [ [SS] AA ] ]
```

DESCRIZIONE

Se si utilizza il programma **'date'** senza argomenti, si ottiene la visualizzazione della data e dell'ora attuale del sistema operativo. Se si indica una sequenza numerica come argomento, si intende invece impostare la data e l'ora del sistema. In tal caso va indicato un numero preciso di cifre, che può essere di otto, dieci o dodici. Se si immettono otto cifre, si sta specificando il mese, il giorno, l'ora e i minuti dell'anno attuale; se si indicano dieci cifre, le ultime due rappresentano l'anno del secolo attuale; se si immettono dodici cifre, l'anno è indicato per esteso nelle ultime quattro cifre.

ESEMPI

```
# date 123123592012 [Invio]
```

Imposta la data di sistema al giorno 31 dicembre 2012, alle ore 23:59.

FILE SORGENTI

'[applic/crt0.mer.s](#)' [[96.1.12](#)]

'[applic/crt0.sep.s](#)' [[96.1.13](#)]

'[applic/date.c](#)' [[96.1.14](#)]

VEDERE ANCHE

time(2) [[87.59](#)], *stime(2)* [[87.59](#)].

86.11 os32: ed(1)



NOME

‘ed’ - creazione e modifica di file di testo

SINTASSI

```
ed
```

DESCRIZIONE

‘ed’ è un programma di creazione e modifica di file di testo che consente di operare su una riga alla volta.

‘ed’ opera in due modalità di funzionamento: comando e inserimento. All’avvio, ‘ed’ si trova in modalità di comando, durante la quale ciò che si inserisce attraverso lo standard input viene interpretato come un comando da eseguire. Per esempio, il comando ‘**1i**’ richiede di passare alla modalità di inserimento, immettendo delle righe a partire dalla prima posizione, spostando quelle presenti in basso. Quando ci si trova in modalità di inserimento, per poter passare alla modalità di comando si introduce un punto isolato, all’inizio di una nuova riga.

Per il momento, in questa pagina di manuale, si omette la descrizione completa dell’utilizzo di ‘ed’.

DIFETTI

Il file che si intende elaborare con ‘ed’ viene caricato o creato completamente nella memoria centrale. Dal momento che os32 consente a ogni processo di gestire una quantità limitata di memoria, si può lavorare soltanto con file di dimensioni ridotte.

AUTORI

Questa edizione di **‘ed’** è stata scritta originariamente da David I. Bell, per **‘sash’** (una shell che include varie funzionalità, da compilare in modo statico). Successivamente, il codice è stato estrapolato da **‘sash’** e reso indipendente, per gli scopi del sistema operativo ELKS (una versione a 16 bit di Linux). Dalla versione estratta per ELKS è stata ottenuta quella di os16, con delle modifiche apportate da Daniele Giacomini, tra cui risulta particolarmente evidente il cambiamento dello stile di impaginazione del codice. Dalla versione per os16 è stata ottenuta quella per os32, senza ulteriori modifiche.

FILE SORGENTI

`‘applic/crt0.mer.s’` [[96.1.12](#)]

`‘applic/crt0.sep.s’` [[96.1.13](#)]

`‘applic/ed.c’` [[96.1.15](#)]

VEDERE ANCHE

shell(1) [[86.24](#)].

86.12 os32: kill(1)

«

NOME

‘kill’ - invio di un segnale ai processi

SINTASSI

```
kill -s nome_segnale pid...
```

```
kill -l
```

DESCRIZIONE

Il programma **'kill'** consente di inviare un segnale, indicato per nome, a uno o più processi, specificati per numero.

OPZIONI

Opzione	Descrizione
-l	Mostra l'elenco dei nomi dei segnali disponibili.
-s <i>nome_segno</i>	Specifica il nome del segnale da inviare ai processi.

NOTE

Non è possibile indicare il segnale per numero, perché lo standard definisce i nomi di un insieme di segnali necessari, ma non stabilisce il numero, il quale può essere attribuito liberamente in fase realizzativa.

DIFETTI

os32 non consente ai processi di attribuire azioni alternative ai segnali; pertanto, si possono ottenere solo quelle predefinite. Tutto quello che si può fare è, eventualmente, bloccare i segnali, esclusi però quelli che non sono mascherabili per loro natura.

FILE SORGENTI

'applic/crt0.mer.s' [[96.1.12](#)]

'applic/crt0.sep.s' [[96.1.13](#)]

'applic/kill.c' [[96.1.20](#)]

VEDERE ANCHE

kill(2) [87.29], *signal(2)* [87.52].

86.13 os32: ln(1)

«

NOME

‘**ln**’ - collegamento dei file

SINTASSI

```
ln file_orig file_nuovo...
```

```
ln file... directory_dest...
```

DESCRIZIONE

‘**ln**’ crea il collegamento fisico di uno o più file. Se l’ultimo argomento è costituito da una directory esistente, si producono collegamenti con gli stessi nomi degli originali, all’interno della directory; se l’ultimo argomento non è una directory già esistente, ci può essere un solo file da collegare, intendendo che si voglia creare un collegamento con quel nome specificato.

Essendo disponibile soltanto la creazione di collegamenti fisici, questi collegamenti possono essere collocati soltanto all’interno del file system di quelli originali, senza contare eventuali innesti ulteriori.

DIFETTI

Non è possibile creare dei collegamenti simbolici, perché os32 non sa gestirli.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [96.1.12]

‘`applic/crt0.sep.s`’ [96.1.13]

‘`applic/ln.c`’ [96.1.21]

VEDERE ANCHE

cp(1) [86.9], *link(2)* [87.30].

86.14 os32: login(1)



NOME

‘**login**’ - inizio di una sessione presso un terminale

SINTASSI

```
login
```

DESCRIZIONE

‘**login**’ richiede l’inserimento di un nominativo-utente e di una parola d’ordine. Questa coppia viene verificata consultando il file ‘`/etc/passwd`’ e se coincide: vengono cambiati i permessi e la proprietà del file di dispositivo del terminale di controllo; viene cambiata la directory corrente in modo da farla coincidere con la directory personale dell’utente; viene avviata la shell, indicata sempre nel file ‘`/etc/passwd`’ per quel tale utente, con i privilegi di questo. La shell, avviata così, va a rimpiazzare il processo di ‘**login**’.

Il programma ‘**login**’ è fatto per essere avviato da ‘**getty**’, non avendo altri utilizzi pratici.

FILE

File	Descrizione
<code>‘/etc/passwd’</code>	Contiene l’elenco degli utenti, con l’associazione al numero UID, alla parola d’ordine necessaria per accedere, alla shell dell’utente. Le altre informazioni eventuali contenute nel file, non sono usate da ‘login’ .

FILE SORGENTI

`‘applic/crt0.mer.s’` [[96.1.12](#)]

`‘applic/crt0.sep.s’` [[96.1.13](#)]

`‘applic/login.c’` [[96.1.22](#)]

VEDERE ANCHE

getty(8) [[92.2](#)], *console(4)* [[89.1](#)].

86.15 os32: ls(1)

«

NOME

‘ls’ - elenco del contenuto delle directory

SINTASSI

```
ls [opzioni] [file] ...
```

DESCRIZIONE

‘ls’ elenca i file e le directory indicati come argomenti della chiamata. Se non vengono indicati file o directory da visualizzare, si ottiene l’elenco del contenuto della directory corrente; inoltre,

questa realizzazione particolare di `ls`, se si indica come argomento solo una directory, ne mostra il contenuto, altrimenti, se gli argomenti sono più di uno, mostra solo i nomi richiesti, eventualmente con le rispettive caratteristiche se è stata usata l'opzione `-l`.

OPZIONI

Opzione	Descrizione
<code>-a</code>	Quando si richiede di mostrare il contenuto di una directory (quella corrente o quella specificata espressamente come primo e unico argomento), con questa opzione si ottiene la visualizzazione anche dei nomi che iniziano con un punto, inclusi <code>.</code> e <code>..</code> .
<code>-l</code>	Con questa opzione si ottiene la visualizzazione di più informazioni sui file e sulle directory elencati.

NOTE

Dal momento che `os32` non considera i gruppi, quando si usa l'opzione `-l`, il nome del gruppo a cui appartiene un file o una directory, non viene visualizzato.

FILE SORGENTI

`'applic/crt0.mer.s'` [[96.1.12](#)]

`'applic/crt0.sep.s'` [[96.1.13](#)]

`'applic/ls.c'` [[96.1.23](#)]

86.16 os32: man(1)

<<

NOME

‘**man**’ - visualizzazione delle pagine di manuale

SINTASSI

```
man [ sezione ] pagina
```

DESCRIZIONE

‘**man**’ visualizza la pagina di manuale indicata come argomento, consentendone lo scorrimento in avanti. La «pagina» viene cercata tra le sezioni, a partire dalla prima. In caso di omonimie tra sezioni differenti, si può specificare il numero della sezione prima del nome della pagina.

Le pagine di manuale di os32 sono semplicemente dei file di testo, collocati nella directory ‘/usr/share/man/’, con nomi del tipo ‘*pagina .n*’, dove *n* è il numero della sezione.

FILE SORGENTI

‘applic/crt0.mer.s’ [[96.1.12](#)]

‘applic/crt0.sep.s’ [[96.1.13](#)]

‘applic/man.c’ [[96.1.24](#)]

VEDERE ANCHE

cat(1) [[86.4](#)].

86.17 os32: mkdir(1)

**NOME**

‘**mkdir**’ - creazione di directory

SINTASSI

```
mkdir [-p] [-m mod_ottale] [directory] ...
```

DESCRIZIONE

‘**mkdir**’ crea una o più directory, corrispondenti ai nomi che costituiscono gli argomenti.

OPZIONI

Opzione	Descrizione
-p	Se la directory che si vuole creare, può richiedere prima la creazione di altre directory, con questa opzione (<i>parents</i>) si generano tutte le directory genitrici necessarie, purché quei nomi non siano già usati per dei file.
-m <i>mod_ottale</i>	Quando si crea una directory, senza specificare questa opzione, si ottengono i permessi 0777 ₈ meno quanto contenuto nella maschera di creazione dei file e delle directory. Con l’opzione ‘-m’ si vanno invece a specificare i permessi desiderati in modo esplicito.

FILE SORGENTI

‘*applic/crt0.mer.s*’ [[96.1.12](#)]

‘*applic/crt0.sep.s*’ [[96.1.13](#)]

‘`applic/mkdir.c`’ [96.1.25]

VEDERE ANCHE

`mkdir(2)` [87.34], `rmdir(2)` [87.41].

86.18 os32: `mmcheck(1)`

«

NOME

‘**mmcheck**’ - verifica della coerenza tra processi e mappa della memoria allocata

SINTASSI

```
mmcheck
```

DESCRIZIONE

Il programma ‘**mmcheck**’ si limita a leggere dai file di dispositivo ‘`/dev/kmem_map`’ e ‘`/dev/kmem_ps`’, per verificare che le informazioni sull’allocazione della memoria relative ai processi in corso siano coerenti con la mappa effettiva.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [96.1.12]

‘`applic/crt0.sep.s`’ [96.1.13]

‘`applic/mmcheck.c`’ [96.1.26]

VEDERE ANCHE

`allocated(1)` [86.2].

86.19 os32: more(1)



NOME

‘**more**’ - visualizzazione di file sullo schermo, permettendo il controllo dello scorrimento dei dati, ma in un solo verso

SINTASSI

```
more file...
```

DESCRIZIONE

‘**more**’ visualizza i file indicati come argomenti della chiamata, sospendendo lo scorrimento del testo dopo alcune righe, consentendo all’utente di decidere come proseguire.

COMANDI

Quando ‘**more**’ sospende lo scorrimento del testo, è possibile introdurre un comando, composto da un solo carattere seguito da [*Invio*], tenendo conto che ciò che non è previsto fa comunque proseguire lo scorrimento:

Comando	Descrizione
n	si richiede di saltare al file successivo, ammesso che ce ne sia un altro;
q	si richiede di interrompere lo scorrimento e di concludere il funzionamento del programma;
Spazio	si richiede di proseguire nella visualizzazione progressiva dei file.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [[96.1.12](#)]

‘applic/crt0.sep.s’ [96.1.13]

‘applic/more.c’ [96.1.27]

VEDERE ANCHE

cat(1) [86.4].

86.20 os32: nc(1)

«

NOME

‘**nc**’ - gestione di connessioni arbitrarie di tipo UDP o TCP

SINTASSI

```
nc [-u] [-l] indirizzo porta
```

DESCRIZIONE

Il programma ‘**nc**’ (noto come «netcat») consente di gestire delle connessioni UDP o TCP, utilizzando il flusso di standard input in trasmissione ed emettendo il flusso ricevuto dalla controparte attraverso lo standard output.

OPZIONI

Opzione	Descrizione
-u	Utilizza il protocollo UDP invece di TCP.
-l	Si mette in ascolto e attende una richiesta di connessione.

Quando non si utilizza l’opzione ‘-l’, l’indirizzo e la porta indicati nella riga di comando, rappresentano la controparte che si intende contattare; se invece si usa l’opzione ‘-l’, si tratta di indirizzo e porta locali.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [96.1.12]

‘`applic/crt0.sep.s`’ [96.1.13]

‘`applic/nc.c`’ [96.1.29]

VEDERE ANCHE

arp(8) [92.1], *ipconfig(8)* [92.5], *route(8)* [92.9], *http(8)* [92.3], *ping(8)* [92.8].

86.21 os32: ps(1)



NOME

‘**ps**’ - visualizzazione dello stato dei processi elaborativi

SINTASSI

```
ps [-u|-g]
```

DESCRIZIONE

‘**ps**’ visualizza l’elenco dei processi, con le informazioni disponibili sul loro stato. L’elenco è provvisto di un’intestazione, come si vede nell’esempio seguente:

pp	p	pg							T * 0x1000	D * 0x1000	stack			
id	id	rp	tty	uid	euid	suid	usage	s	addr	size	addr	size	usage	name
0	0	0	0000	0	0	0	00.02	r	00000	028e	00000	0000	0%	os32 kernel
0	1	0	0000	0	0	0	00.00	s	0051e	003a	00000	0000	9%	/bin/init
1	2	2	0500	1001	1001	1001	00.00	s	0033d	003a	00000	0000	26%	/bin/shell
1	3	3	0501	0	0	0	00.00	s	0028f	003a	00000	0000	36%	/bin/login
1	4	4	0502	0	0	0	00.00	s	002c9	003a	00000	0000	36%	/bin/login
1	5	5	0503	0	0	0	00.00	s	00303	003a	00000	0000	36%	/bin/login
2	6	2	0500	1001	1001	1001	00.00	R	003b1	003a	00000	0000	20%	/bin/ps

La prima colonna, con la sigla «ppid», ovvero *parent pid*, riporta il numero del processo genitore; la seconda, con la sigla «pid», *process id*, indica il numero del processo preso in considerazione; la terza, con la sigla «pgrp», *process group*, indica il gruppo a cui appartiene il processo; la quarta colonna, «tty», indica il terminale associato, ammesso che ci sia, come numero di dispositivo, ma in base sedici. Le colonne «uid», «euid» e «suid», riguardano l'identità dell'utente, per conto del quale sono in funzione i processi, rappresentando, nell'ordine, l'identità reale (*real user id*), quella efficace (*effective user id*) e quella salvata precedentemente (*saved user id*).

La colonna «usage» indica il tempo di utilizzo della CPU; la colonna «s» indica lo stato del processo, il cui significato può essere interpretato con l'aiuto della tabella successiva:

Lettera	Significato	Descrizione
R	<i>running</i>	in corso di esecuzione
r	<i>ready</i>	pronto per essere messo in esecuzione
s	<i>sleeping</i>	in attesa
z	<i>zombie</i>	terminato e non più in memoria, per il quale si attende di passare lo stato di uscita al processo genitore.

Le prime due colonne «addr» e «size» indicano l'indirizzo iniziale e l'estensione dell'area codice del processo, in memoria (*text*); le altre due successive indicano l'indirizzo iniziale e l'estensione dell'area dati del processo, in memoria (*data*). Gli indirizzi e le dimensioni delle aree di memoria utilizzate appaiono divisi per 1000_{16} , ovvero sono multipli di 4096 byte. La colonna «stack

usage» indica la percentuale di utilizzo dello spazio attribuito alla pila dei dati (*stack pointer*).

L'ultima colonna indica il nome del programma, assieme al suo percorso, con il quale il processo è stato avviato.

OPZIONI

Opzione	Descrizione
-u	Nell'elenco mostra i numeri UID relativi al processo. Questa opzione è predefinita.
-g	Nell'elenco, al posto dei numeri UID, mostra i numeri GID.

NOTE

L'elenco dei processi include anche il kernel, il quale occupa correttamente la prima posizione (processo zero).

FILE

'**ps**' trae le informazioni sullo stato dei processi da un file di dispositivo speciale: '/dev/kmem_ps'.

FILE SORGENTI

'applic/crt0.mer.s' [96.1.12]

'applic/crt0.sep.s' [96.1.13]

'applic/ps.c' [96.1.31]

86.22 os32: rm(1)

NOME

'**rm**' - cancellazione di file

SINTASSI

```
rm file...
```

DESCRIZIONE

‘**rm**’ consente di cancellare i file indicati come argomento.

DIFETTI

Non è possibile eseguire la cancellazione ricorsiva di una directory.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [[96.1.12](#)]

‘`applic/crt0.sep.s`’ [[96.1.13](#)]

‘`applic/rm.c`’ [[96.1.32](#)]

VEDERE ANCHE

`unlink(2)` [[87.62](#)].

86.23 os32: `rmdir(1)`

«

NOME

‘**rmdir**’ - cancellazione di directory vuote

SINTASSI

```
rmdir directory...
```

DESCRIZIONE

Il programma ‘**rmdir**’ consente di cancellare le directory indicate come argomento, purché queste siano vuote.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [96.1.12]

‘`applic/crt0.sep.s`’ [96.1.13]

‘`applic/rmdir.c`’ [96.1.33]

VEDERE ANCHE

unlink(2) [87.62].

86.24 os32: shell(1)



NOME

‘**shell**’ - interprete dei comandi

SINTASSI

```
shell
```

DESCRIZIONE

‘**shell**’ è l’interprete dei comandi di os32. Di norma viene avviato da ‘**login**’, in base alla configurazione contenuta nel file ‘`/etc/passwd`’.

‘**shell**’ interpreta i comandi inseriti; se si tratta di un comando interno lo esegue direttamente, altrimenti cerca e avvia un programma con il nome corrispondente, rimanendo in attesa fino alla conclusione del processo relativo, per riprendere poi il controllo.

DIFETTI

L’interpretazione della riga di comando è letterale, pertanto non c’è alcuna espansione di caratteri speciali, variabili di ambiente o altro; inoltre, non è possibile eseguire script.

A volte, quando un processo avviato da **'shell'** termina di funzionare, il processo di **'shell'** non viene risvegliato correttamente, rendendo inutilizzabile il terminale. Per ovviare all'inconveniente, si può premere la combinazione [*Ctrl c*], con la quale viene inviato il segnale **'SIGINT'** a tutti i processi del gruppo associato al terminale.

Anche il fatto che un segnale generato con una combinazione di tasti si trasmetta a tutti i processi del gruppo associato al terminale è un'anomalia, tuttavia fa parte delle particolarità dovute alla semplificazione di os32.

FILE SORGENTI

'*applic/crt0.mer.s*' [96.1.12]

'*applic/crt0.sep.s*' [96.1.13]

'*applic/shell.c*' [96.1.35]

VEDERE ANCHE

input_line(3) [88.68].

86.25 os32: *t_(1)*

«

NOME

'*t_...*' - programmi di prova

DESCRIZIONE

I programmi con prefisso '*t_...*' sono delle prove per verificare alcune funzionalità di os32.

86.26 os32: touch(1)



NOME

‘**touch**’ - creazione di un file vuoto oppure aggiornamento della data di modifica

SINTASSI

```
touch file...
```

DESCRIZIONE

‘**touch**’ crea dei file vuoti, se quelli indicati come argomento non sono esistenti; altrimenti, aggiorna le date di accesso e di modifica, sulla base dello stato dell’orologio di sistema.

DIFETTI

Non è possibile attribuire una data arbitraria; inoltre, a causa della limitazione del tipo di file system utilizzato, non è possibile distinguere tra date di accesso e modifica dei file.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [[96.1.12](#)]

‘`applic/crt0.sep.s`’ [[96.1.13](#)]

‘`applic/touch.c`’ [[96.1.51](#)]

86.27 os32: tty(1)



NOME

‘**tty**’ - nome del file di dispositivo del terminale associato allo standard input

SINTASSI

```
tty
```

DESCRIZIONE

Il programma **'tty'** individua il dispositivo del terminale associato allo standard input e lo traduce in un percorso che descrive il file di dispositivo corrispondente (ovvero il file di dispositivo che dovrebbe corrispondergli)

FILE SORGENTI

'[applic/crt0.mer.s](#)' [[96.1.12](#)]

'[applic/crt0.sep.s](#)' [[96.1.13](#)]

'[applic/tty.c](#)' [[96.1.52](#)]

86.28 os32: yes(1)

«

NOME

'yes' - visualizzazione ripetuta di un testo

SINTASSI

```
yes [ testo ]
```

DESCRIZIONE

Il programma **'yes'** emette sullo schermo, all'infinito, il contenuto del testo fornito come argomento, oppure la lettera **'y'**. In tutti i casi, il testo o la lettera sono seguiti dal codice di interruzione di riga (*new line*).

FILE SORGENTI

`'applic/crt0.mer.s'` [[96.1.12](#)]

`'applic/crt0.sep.s'` [[96.1.13](#)]

`'applic/yes.c'` [[96.1.54](#)]

Sezione 2: chiamate di sistema

87.1 os32: `_Exit(2)`

Vedere `_exit(2)` [87.2].

87.2 os32: `_exit(2)`

NOME

`'_exit'`, `'_Exit'` - conclusione del processo chiamante

SINTASSI

```
#include <unistd.h>
void _exit (int status);
```

```
#include <stdlib.h>
void _Exit (int status);
```

DESCRIZIONE

Le funzioni `_exit()` e `_Exit()` sono equivalenti e servono per concludere il processo chiamante, con un valore pari a quello indicato come argomento (*status*), purché inferiore o uguale 255 (FF₁₆).

La conclusione del processo implica anche la chiusura dei suoi file aperti, e l'affidamento di eventuali processi figli al processo numero uno (`'init'`); inoltre, si ottiene l'invio di un segnale SIGCHLD al processo genitore di quello che viene concluso.

VALORE RESTITUITO

La funzione non può restituire alcunché, dal momento che la sua esecuzione comporta la morte del processo.

FILE SORGENTI

‘lib/unistd.h’ [95.30]
‘lib/unistd/_exit.c’ [95.30.1]
‘lib/stdlib.h’ [95.19]
‘lib/stdlib/_Exit.c’ [95.19.1]
‘lib/sys/os32/sys.s’ [95.21.7]
‘kernel/ibm_i386/isr.s’ [94.6.21]
‘kernel/proc/sysroutine.c’ [94.14.28]
‘kernel/lib_s/s__exit.c’ [94.8.1]

VEDERE ANCHE

execve(2) [87.14], *fork(2)* [87.19], *kill(2)* [87.29], *wait(2)* [87.63], *atexit(3)* [88.7], *exit(3)* [88.7].

87.3 os32: accept(2)

«

NOME

‘**accept**’ - accetta una richiesta di connessione in un socket

SINTASSI

```
#include <sys/socket.h>
int accept (int sfdn, struct sockaddr *addr,
           socklen_t *addrlen);
```

DESCRIZIONE

La funzione di sistema *accept()* viene usata in os32 solo in relazione a socket di tipo *SOCK_STREAM*, per il protocollo TCP, in ascolto in attesa di connessione. Se il descrittore *sfdn* corrisponde a queste caratteristiche, la funzione estrae la prima richiesta in attesa, crea una nuova connessione e restituisce il numero del nuovo descrittore relativo.

Perché la funzione possa svolgere il proprio compito, è necessario che *sfdn* sia il descrittore di un socket TCP creato con la funzione *socket()* [87.54], collegato a una porta locale con la funzione *bind()* [87.4] e in ascolto di richieste di connessione attraverso la funzione *listen()* [87.31].

Per poter utilizzare la funzione *accept()* è necessario predisporre una memoria tampone in cui collocare una variabile strutturata di tipo ‘**struct sockaddr**’, a cui punta *addr*. Questa struttura viene popolata da *accept()* con l’indirizzo della controparte che richiede di connettersi (il tipo di indirizzo, l’indirizzo IPv4 e la porta).

Il parametro *addrlen* deve puntare a una variabile che contiene inizialmente la dimensione massima di **addr*; la funzione *accept()* si limita a utilizzare al massimo lo spazio così indicato, ma il valore di **addrlen* viene comunque aggiornato alla dimensione effettiva che ha o che dovrebbe avere la struttura completa. Pertanto, al termine del funzionamento di *accept()*, se il valore che **addrlen* dovesse essere maggiore di quello indicato inizialmente, vuol dire che la struttura ottenuta è incompleta.

Se si usa *accept()* quando non ci sono richieste di connessione in attesa, questa blocca il funzionamento del processo chiamante,

fino a che si ottiene effettivamente una richiesta. Tuttavia, prima di usare la funzione è possibile attribuire al descrittore l'opzione ***O_NONBLOCK*** e in tal modo ottenere che la funzione termini ugualmente il proprio lavoro restituendo un errore di tipo ***EAGAIN***.

VALORE RESTITUITO

In caso di successo la funzione restituisce un valore non negativo, corrispondente al descrittore del socket creato per la nuova connessione. In presenza di errori, la funzione restituisce -1 e aggiorna la variabile ***errno***.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore <i>sfdn</i> non è valido; oppure, la richiesta in coda non corrisponde a un descrittore di file.
ENOTSOCK	Il descrittore <i>sfdn</i> non è un socket; oppure, la richiesta in coda non corrisponde a un socket.
EOPNOTSUPP	Il descrittore <i>sfdn</i> non è un socket di tipo <i>SOCK_STREAM</i> , oppure il protocollo relativo non è di tipo <i>IPPROTO_TCP</i> .
EINVAL	Il descrittore <i>sfdn</i> non è un socket in ascolto (la connessione non è nello stato <i>TCP_LISTEN</i>).
EAGAIN	Non ci sono richieste di connessione in coda, ma è possibile ritentare.

FILE SORGENTI

'lib/sys/socket.h' [95.23]

'lib/sys/socket/accept.c' [95.23.1]

'kernel/lib_s/s_accept.c' [94.8.2]

VEDERE ANCHE

bind(2) [87.4], *connect(2)* [87.11], *listen(2)* [87.31], *socket(2)* [87.54].

87.4 os32: bind(2)



NOME

'**bind**' - associa un indirizzo e una porta locali a un socket

SINTASSI

```
#include <sys/socket.h>
int bind (int sfdn, const struct sockaddr *addr,
          socklen_t addrlen);
```

DESCRIZIONE

Dopo che un socket è stato creato con l'ausilio della funzione *socket()* [87.54], questo contiene soltanto l'informazione del tipo (nel caso di os32 può trattarsi soltanto di *AF_INET*), senza indirizzo e porta. Per attribuire tali informazioni, riferite al lato locale della connessione, si può usare la funzione *bind()*.

Per os32 la variabile strutturata che fa capo a **addr* può contenere solo informazioni relative a IPv4.

VALORE RESTITUITO

In caso di successo la funzione restituisce zero; in caso di errore si ottiene invece -1 e l'aggiornamento della variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore <i>sfdn</i> non è valido.
ENOTSOCK	Il descrittore <i>sfdn</i> non è un socket.
EINVAL	I dati contenuti in <i>*addr</i> non sono validi.
EADDRNOTAVAIL	All'interno di <i>*addr</i> manca l'indicazione della porta locale.
EACCES	Si sta tentando di aprire una porta locale minore di 1024, disponendo però di un UID efficace diverso da zero.
EAFNOSUPPORT	Il tipo di indirizzamento del socket è diverso da <i>AF_INET</i> .
EOPNOTSUPP	Il descrittore <i>sfdn</i> non è un socket di tipo <i>SOCK_STREAM</i> , oppure il protocollo relativo non è di tipo <i>IPPROTO_TCP</i> e nemmeno <i>IPPROTO_UDP</i> .
EAGAIN	Non ci sono porte disponibili.

FILE SORGENTI

'lib/sys/socket.h' [95.23]

'lib/sys/socket/bind.c' [95.23.2]

'kernel/lib_s/s_bind.c' [94.8.3]

VEDERE ANCHE

accept(2) [87.3], *connect(2)* [87.11], *listen(2)* [87.31], *socket(2)* [87.54].

87.5 os32: brk(2)



NOME

‘**brk**’, ‘**sbrk**’ - modifica della dimensione del segmento dati

SINTASSI

```
#include <unistd.h>
int brk (void *address);
void *sbrk (intptr_t increment);
```

DESCRIZIONE

Le funzione *brk()* e *sbrk()* permettono di modificare la dimensione del segmento dati del processo, generalmente per aumentarlo. Per os32, l’area di memoria che può essere espansa o contratta, si trova dopo la pila dei dati. L’incremento del segmento dati usato da un processo implica generalmente la copia dello stesso in una nuova posizione.

La funzione *brk()* permette di richiedere di fissare la fine del segmento dati in corrispondenza dell’indirizzo fornito come argomento, in qualità di puntatore generico.

La funzione *sbrk()* permette di richiedere la modifica del segmento dati specificandone l’incremento e restituendo il valore precedente della fine del segmento dati, in forma di puntatore. In tal modo, la funzione *sbrk()* può essere usata anche solo per conoscere la fine attuale del segmento dati, fornendo come argomento un incremento pari a zero.

VALORE RESTITUITO

La funzione *brk()* restituisce zero se l'operazione richiesta è completata con successo, diversamente restituisce -1 , aggiornando la variabile *errno*.

La funzione *sbrk()* restituisce un puntatore valido, se l'operazione si conclude con successo, diversamente si ottiene l'equivalente del valore -1 , tradotto in forma di puntatore generico, con il contestuale aggiornamento della variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
ENOMEM	Memoria insufficiente per completare la richiesta.
EINVAL	L'argomento fornito non è valido.

NOTE

Le funzioni *brk()* e *sbrk()* servono per l'allocazione dinamica della memoria attraverso *malloc()*, *free()* e *realloc()*. Pertanto, è meglio avvalersi di queste ultime, piuttosto di rischiare di mettere in conflitto le due cose.

FILE SORGENTI

'lib/unistd.h' [95.30]

'lib/unistd/brk.c' [95.30.3]

'lib/unistd/sbrk.c' [95.30.34]

'lib/sys/os32/sys.s' [95.21.7]

'kernel/ibm_i386/isr.s' [94.6.21]

'kernel/proc/sysroutine.c' [94.14.28]

‘kernel/lib_s/s_brk.c’ [94.8.4]

‘kernel/lib_s/s_sbrk.c’ [94.8.33]

VEDERE ANCHE

free(3) [88.76], *malloc(3)* [88.76], *realloc(3)* [88.76].

87.6 os32: chdir(2)



NOME

‘**chdir**’, ‘**fchdir**’ - modifica della directory corrente

SINTASSI

```
#include <unistd.h>
int chdir (const char *path);
int chdir (int fdn);
```

DESCRIZIONE

La funzione *chdir()* cambia la directory corrente, in modo che quella nuova corrisponda al percorso annotato nella stringa *path*. La funzione *fchdir()* dovrebbe svolgere lo stesso compito, indicando la nuova directory come descrittore già aperto; tuttavia os32 gestisce le directory esclusivamente in forma di percorso.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Argomento non valido.
EACCES	Accesso negato.
ENOTDIR	Uno dei componenti del percorso non è una directory.
ENOENT	Uno dei componenti del percorso non esiste.
E_NOT_IMPLEMENTED	La funzione <i>fchdir()</i> di os32 non può essere realizzata, pertanto risponde con questo errore.

FILE SORGENTI

‘lib/unistd.h’ [95.30]

‘lib/unistd/chdir.c’ [95.30.4]

‘lib/unistd/fchdir.c’ [95.30.16]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

‘kernel/lib_s/s_chdir.c’ [94.8.5]

VEDERE ANCHE

rmdir(2) [87.41], *access(3)* [88.4].

87.7 os32: chmod(2)



NOME

‘**chmod**’, ‘**fchmod**’ - cambiamento della modalità dei permessi di un file

SINTASSI

```
#include <sys/stat.h>
int chmod (const char *path, mode_t mode);
int fchmod (int fdn, mode_t mode);
```

DESCRIZIONE

Le funzioni *chmod()* e *fchmod()* consentono di modificare la modalità dei permessi di accesso di un file. La funzione *chmod()* individua il file su cui intervenire attraverso un percorso, ovvero la stringa *path*; la funzione *fchmod()*, invece, richiede l’indicazione del numero di un descrittore di file, già aperto. In entrambi i casi, l’ultimo argomento serve a specificare la nuova modalità dei permessi.

Tradizionalmente, i permessi si scrivono attraverso un numero in base otto; in alternativa, si possono usare convenientemente della macro-variabili, dichiarate nel file ‘`sys/stat.h`’, combinate assieme con l’operatore binario OR.

Modalità simbolica	Modalità numerica	Descrizione
S_IRWXU	00700 ₈	Lettura, scrittura ed esecuzione per l'utente proprietario.
S_IRUSR	00400 ₈	Lettura per l'utente proprietario.
S_IWUSR	00200 ₈	Scrittura per l'utente proprietario.
S_IXUSR	00100 ₈	Esecuzione per l'utente proprietario.
S_IRWXG	00070 ₈	Lettura, scrittura ed esecuzione per il gruppo.
S_IRGRP	00040 ₈	Lettura per il gruppo.
S_IWGRP	00020 ₈	Scrittura per il gruppo.
S_IXGRP	00010 ₈	Esecuzione per il gruppo.
S_IRWXO	00007 ₈	Lettura, scrittura ed esecuzione per gli altri utenti.
S_IROTH	00004 ₈	Lettura per gli altri utenti.
S_IWOTH	00002 ₈	Scrittura per gli altri utenti.
S_IXOTH	00001 ₈	Esecuzione per gli altri utenti.

os32 non considera i permessi SUID (*Set user id*), SGID (*Set group id*) e Sticky, che nella tabella non sono stati nemmeno annotati; inoltre, non tiene in considerazione i permessi legati al gruppo.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EACCES	Permesso negato.
EBADF	Il descrittore del file richiesto non è valido.

FILE SORGENTI

'lib/sys/stat.h' [[95.25](#)]

'lib/sys/stat/chmod.c' [[95.25.1](#)]

'lib/sys/stat/fchmod.c' [[95.25.2](#)]

'lib/sys/os32/sys.s' [[95.21.7](#)]

'kernel/ibm_i386/isr.s' [[94.6.21](#)]

'kernel/proc/sysroutine.c' [[94.14.28](#)]

'kernel/lib_s/s_chmod.c' [[94.8.6](#)]

'kernel/lib_s/s_fchmod.c' [[94.8.13](#)]

VEDERE ANCHE

chmod(1) [[86.7](#)], *chown(2)* [[87.8](#)], *open(2)* [[87.37](#)], *stat(2)* [[87.55](#)].

87.8 os32: `chown(2)`**NOME**

‘**chown**’, ‘**fchown**’ - modifica della proprietà dei file

SINTASSI

```
#include <unistd.h>
int chown (const char *path, uid_t uid, gid_t gid);
int fchown (int fdn, uid_t uid, gid_t gid);
```

DESCRIZIONE

Le funzioni *chown()* e *fchown()*, modificano la proprietà di un file, fornendo il numero UID e il numero GID. Il file viene indicato, rispettivamente, attraverso il percorso scritto in una stringa, oppure come numero di descrittore già aperto.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Argomento non valido.
EPERM	Permessi insufficienti per eseguire l'operazione.
ENOTDIR	Uno dei componenti del percorso non è una directory.
ENOENT	Uno dei componenti del percorso non esiste.
EBADF	Il descrittore del file non è valido.

DIFETTI

Le funzioni consentono di attribuire il numero del gruppo, ma os32 non valuta i permessi di accesso ai file, relativi ai gruppi.

FILE SORGENTI

'lib/unistd.h' [[95.30](#)]

'lib/unistd/chown.c' [[95.30.5](#)]

'lib/sys/os32/sys.s' [[95.21.7](#)]

'kernel/ibm_i386/isr.s' [[94.6.21](#)]

'kernel/proc/sysroutine.c' [[94.14.28](#)]

'kernel/lib_s/s_chown.c' [[94.8.7](#)]

'kernel/lib_s/s_fchown.c' [[94.8.14](#)]

VEDERE ANCHE

chmod(2) [[87.7](#)].

87.9 os32: clock(2)

<<

NOME

‘**clock**’ - tempo della CPU espresso in unità ‘**clock_t**’

SINTASSI

```
#include <time.h>
clock_t clock (void);
```

DESCRIZIONE

La funzione *clock()* restituisce il tempo di utilizzo della CPU, espresso in unità ‘**clock_t**’, corrispondenti a secondi diviso il valore della macro-variabile **CLOCKS_PER_SEC**. Per os32, come dichiarato nel file ‘time.h’, il valore di **CLOCKS_PER_SEC** è 100, essendo la frequenza del temporizzatore interno regolata a 100 Hz.

VALORE RESTITUITO

La funzione restituisce il tempo di CPU, espresso in centesimi di secondo.

FILE SORGENTI

‘lib/time.h’ [95.29]

‘lib/time/clock.c’ [95.29.2]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

‘kernel/lib_s/s_clock.c’ [94.8.8]

VEDERE ANCHE*time(2)* [87.59].

87.10 os32: close(2)

NOME‘**close**’ - chiusura di un descrittore di file**SINTASSI**

```
#include <unistd.h>
int close (int fdn);
```

DESCRIZIONE

Le funzioni *close()* chiude un descrittore di file.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore del file non è valido.

FILE SORGENTI

‘lib/unistd.h’ [95.30]

‘lib/unistd/close.c’ [95.30.6]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

‘kernel/lib_s/s_close.c’ [94.8.9]

VEDERE ANCHE

fcntl(2) [87.18], *open(2)* [87.37], *fclose(3)* [88.28].

87.11 os32: connect(2)

«

NOME

‘**connect**’ - inizia una connessione su un socket

SINTASSI

```
#include <sys/socket.h>
int connect (int sfdn, const struct sockaddr *addr,
             socklen_t addrlen);
```

DESCRIZIONE

Un socket che sia già stato associato a una porta locale, può essere collegato a un socket remoto attraverso la funzione *connect()*. Si distinguono due casi: se il protocollo è «connesso» (*IPPROTO_TCP*), questo procedimento può essere eseguito una volta sola, se invece non lo è (*IPPROTO_UDP*), ci si può connettere successivamente a socket remoti differenti.

Si intende, pertanto, che **addr* si riferisca all’indirizzo del socket remoto.

VALORE RESTITUITO

In caso di successo la funzione restituisce zero; in caso di errore si ottiene invece -1 e l’aggiornamento della variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore <i>sfdn</i> non è valido.
ENOTSOCK	Il descrittore <i>sfdn</i> non è un socket.
EINVAL	I dati contenuti in <i>*addr</i> non sono validi.
EADDRNOTAVAIL	All'interno di <i>*addr</i> manca l'indicazione dell'indirizzo o della porta da raggiungere.
EAGAIN	La porta locale del socket <i>sfdn</i> non risulta specificata, e inoltre non è possibile attribuirne una in modo automatico.
EISCONN	Esiste già una connessione TCP in corso con il socket <i>sfdn</i> e si sta tentando di cambiarne i parametri.
EAFNOSUPPORT	Il tipo di indirizzamento del socket <i>sfdn</i> è diverso da <i>AF_INET</i> .

FILE SORGENTI

'lib/sys/socket.h' [95.23]

'lib/sys/socket/connect.c' [95.23.3]

'kernel/lib_s/s_connect.c' [94.8.10]

VEDERE ANCHE

accept(2) [87.3], *bind(2)* [87.4], *listen(2)* [87.31], *socket(2)* [87.54].

87.12 os32: dup(2)

**NOME**

‘dup’, ‘dup2’ - duplicazione di descrittori di file

SINTASSI

```
#include <unistd.h>
int dup (int fdn_old);
int dup2 (int fdn_old, int fdn_new);
```

DESCRIZIONE

Le funzioni *dup()* e *dup2()* servono a duplicare un descrittore di file. La funzione *dup()* duplica il descrittore *fdn_old*, utilizzando il numero di descrittore libero più basso che sia disponibile; *dup2()*, invece, richiede che il nuovo numero di descrittore sia specificato, attraverso il parametro *fdn_new*. Tuttavia, se il numero di descrittore *fdn_new* risulta utilizzato, questo viene chiuso prima di diventare la copia di *fdn_old*.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Uno dei descrittori specificati non è valido.
EMFILE	Troppi file aperti per il processo.

FILE SORGENTI

‘lib/unistd.h’ [95.30]
‘lib/unistd/dup.c’ [95.30.7]
‘lib/unistd/dup2.c’ [95.30.8]
‘lib/sys/os32/sys.s’ [95.21.7]
‘kernel/ibm_i386/isr.s’ [94.6.21]
‘kernel/proc/sysroutine.c’ [94.14.28]
‘kernel/fs/fd_dup.c’ [94.5.1]
‘kernel/lib_s/s_dup.c’ [94.8.11]
‘kernel/lib_s/s_dup2.c’ [94.8.12]

VEDERE ANCHE

close(2) [87.10], *fcntl(2)* [87.18], *open(2)* [87.37].

87.13 os32: dup2(2)

Vedere *dup(2)* [87.12].

87.14 os32: execve(2)

NOME

‘**execve**’ - esecuzione di un file

SINTASSI

```
#include <unistd.h>
int execve (const char *path, char *const argv[],
           char *const envp[]);
```

DESCRIZIONE

La funzione *execve()* è quella che avvia effettivamente un programma, mentre le altre funzioni ‘*exec...()*’ offrono semplicemente un’interfaccia differente per l’avvio, ma poi si avvalgono di *execve()* per svolgere effettivamente quanto loro richiesto.

La funzione *execve()* avvia il file il cui percorso è specificato come stringa, nel primo argomento.

Il secondo argomento deve essere un array di stringhe, dove la prima deve rappresentare il nome del programma avviato e le successive sono gli argomenti da passare al programma. L’ultimo elemento di tale array deve essere un puntatore nullo, per poter essere riconosciuto.

Il terzo argomento deve essere un array di stringhe, rappresentanti l’ambiente da passare al nuovo processo. Per ambiente si intende l’insieme delle variabili di ambiente, pertanto queste stringhe devono avere la forma ‘*nome=valore*’, per essere riconoscibili. Anche in questo caso, per poter individuare l’ultimo elemento dell’array, questo deve essere un puntatore nullo.

VALORE RESTITUITO

Se *execve()* riesce nel suo compito, non può restituire alcunché, dato che in quel momento, il processo chiamante viene rimpiazzato da quello del file che viene eseguito. Pertanto, se viene restituito qualcosa, può trattarsi solo di un valore che rappresenta un errore, ovvero -1 , aggiornando anche la variabile *errno* di conseguenza.

ERRORI

Valore di <i>errno</i>	Significato
E2BIG	Ci sono troppi argomenti.
ENOMEM	Memoria insufficiente.
ENOENT	Il file richiesto non esiste.
EACCES	Il file non può essere avviato per la mancanza dei permessi di accesso necessari.
ENOEXEC	Il file non può essere un file eseguibile, perché non ne ha le caratteristiche.
EIO	Errore di input-output.

DIFETTI

os32 non prevede l'interpretazione di script, perché non esiste alcun programma in grado di farlo. Anche la shell di os32 si limita a eseguire i comandi inseriti, ma non può interpretare un file.

FILE SORGENTI

'lib/unistd.h' [[95.30](#)]

'lib/unistd/execl.c' [[95.30.11](#)]

'lib/sys/os32/sys.s' [[95.21.7](#)]

'kernel/ibm_i386/isr.s' [[94.6.21](#)]

'kernel/proc/sysroutine.c' [[94.14.28](#)]

'kernel/proc/proc_sys_exec.c' [[94.14.22](#)]

VEDERE ANCHE

fork(2) [87.19], *exec(3)* [88.21], *getopt(3)* [88.56], *environ(7)* [91.1].

87.15 os32: *fchdir(2)*

« Vedere *chdir(2)* [87.6].

87.16 os32: *fchmod(2)*

« Vedere *chmod(2)* [87.7].

87.17 os32: *fchown(2)*

« Vedere *chown(2)* [87.8].

87.18 os32: *fcntl(2)*

«

NOME

‘**fcntl**’ - configurazione e intervento sui descrittori di file

SINTASSI

```
#include <fcntl.h>
int fcntl (int fdn, int cmd, ...);
```

DESCRIZIONE

La funzione *fcntl()* esegue un’operazione, definita dal parametro *cmd*, sul descrittore richiesto come primo parametro (*fdn*). A seconda del tipo di operazione richiesta, potrebbero essere necessari degli argomenti ulteriori, i quali però non possono essere

formalizzati in modo esatto nel prototipo della funzione. Il valore del secondo parametro che rappresenta l'operazione richiesta, va fornito in forma di costante simbolica, come descritto nell'elenco seguente.

Sintassi	Descrizione
<pre>fcntl (<i>fdn</i>, F_DUPFD, (int) <i>fdn_min</i>)</pre>	<p>Richiede la duplicazione del descrittore di file <i>fdn</i>, in modo tale che la copia abbia il numero di descrittore minore possibile, ma maggiore o uguale a quello indicato come argomento <i>fdn_min</i>.</p>
<pre>fcntl (<i>fdn</i>, F_GETFD) fcntl (<i>fdn</i>, F_SETFD, (int) <i>fd_flags</i>)</pre>	<p>Rispettivamente, legge o imposta, gli indicatori del descrittore di file <i>fdn</i> (eventualmente noti come <i>file descriptor flags</i> o solo <i>fd flags</i>). Per il momento, è possibile impostare un solo indicatore, ‘FD_CLOEXEC’, pertanto, al posto di <i>fd_flags</i> si può mettere solo la costante ‘FD_CLOEXEC’.</p>
<pre>fcntl (<i>fdn</i>, F_GETFL) fcntl (<i>fdn</i>, F_SETFL, (int) <i>fl_flags</i>)</pre>	<p>Rispettivamente, legge o imposta, gli indicatori dello stato del file, relativi al descrittore <i>fdn</i> (eventualmente noti come <i>file flaga</i> o solo <i>fl flags</i>). Per impostare questi indicatori, vanno combinate delle costanti simboliche: ‘O_RDONLY’, ‘O_WRONLY’, ‘O_RDWR’, ‘O_CREAT’, ‘O_EXCL’, ‘O_NOCTTY’, ‘O_TRUNC’.</p>

VALORE RESTITUITO

Il significato del valore restituito dalla funzione dipende dal tipo di operazione richiesta, come sintetizzato dalla tabella successiva.

Operazione richiesta	Significato del valore restituito
F_DUPFD	Si ottiene il numero del descrittore prodotto dalla copia, oppure -1 in caso di errore.
F_GETFD	Si ottiene il valore degli indicatori del descrittore (<i>fd flags</i>), oppure -1 in caso di errore.
F_GETFL	Si ottiene il valore degli indicatori del file (<i>fl flags</i>), oppure -1 in caso di errore.
F_GETOWN F_SETOWN F_GETLK F_SETLK F_SETLKW	Si ottiene -1, in quanto si tratta di operazioni non realizzate in questa versione della funzione, per os32.
altri tipi di operazione	Si ottiene 0 in caso di successo, oppure -1 in caso di errore.

ERRORI

Valore di <i>errno</i>	Significato
E_NOT_IMPLEMENTED	È stato richiesto un tipo di operazione che non è disponibile nel caso particolare di os32.
EINVAL	È stato richiesto un tipo di operazione non valido.

FILE SORGENTI

'lib/fcntl.h' [95.6]

'lib/fcntl/fcntl.c' [95.6.2]

'lib/sys/os32/sys.s' [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

‘kernel/lib_s/s_fcntl.c’ [94.8.15]

VEDERE ANCHE

dup(2) [87.12], *dup2(2)* [87.12], *open(2)* [87.37].

87.19 os32: fork(2)

«

NOME

‘**fork**’ - sdoppiamento di un processo, ovvero creazione di un processo figlio

SINTASSI

```
#include <unistd.h>
pid_t fork (void);
```

DESCRIZIONE

La funzione *fork()* crea una copia del processo in corso, la quale copia diventa un processo figlio del primo. Il processo figlio eredita una copia dei descrittori di file aperti e di conseguenza dei flussi di file e directory.

Il processo genitore riceve dalla funzione il valore del numero PID del processo figlio avviato; il processo figlio si mette a funzionare dal punto in cui si trova la funzione *fork()*, restituendo però un valore nullo: in questo modo tale processo figlio può riconoscersi come tale.

VALORE RESTITUITO

La funzione restituisce al processo genitore il numero PID del processo figlio; al processo figlio restituisce zero. In caso di problemi, invece, il valore restituito è -1 e la variabile *errno* risulta aggiornata di conseguenza.

ERRORI

Valore di <i>errno</i>	Significato
ENOMEM	Memoria insufficiente per avviare un altro processo.

FILE SORGENTI

‘lib/unistd.h’ [95.30]

‘lib/unistd/fork.c’ [95.30.18]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

‘kernel/lib_s/s_fork.c’ [94.8.16]

VEDERE ANCHE

execve(2) [87.14], *wait(2)* [87.63], *exec(3)* [88.21].

87.20 os32: *fstat(2)*

Vedere *stat(2)* [87.55].

87.21 os32: getcwd(2)

**NOME**

‘**getcwd**’ - determinazione della directory corrente

SINTASSI

```
#include <unistd.h>
char *getcwd (char *buffer, size_t size);
```

DESCRIZIONE

La funzione *getcwd()* modifica il contenuto dell’area di memoria a cui punta *buffer*, copiandovi al suo interno la stringa che rappresenta il percorso della directory corrente. La scrittura all’interno di *buffer* può prolungarsi al massimo per *size* byte, incluso il codice nullo di terminazione delle stringhe.

VALORE RESTITUITO

La funzione restituisce il puntatore alla stringa che rappresenta il percorso della directory corrente, il quale deve coincidere con *buffer*. In caso di errore, invece, la funzione restituisce il puntatore nullo ‘**NULL**’.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il puntatore <i>buffer</i> non è valido.
E_LIMIT	Il percorso della directory corrente è troppo lungo, rispetto ai limiti realizzativi di os32.

DIFETTI

La funzione *getcwd()* di os32 deve comunicare con il kernel per ottenere l'informazione che le serve, perché la «u-area» (*User area*) è trattenuta all'interno del kernel stesso.

FILE SORGENTI

'lib/unistd.h' [95.30]

'lib/unistd/getcwd.c' [95.30.19]

'lib/sys/os32/sys.s' [95.21.7]

'kernel/ibm_i386/isr.s' [94.6.21]

'kernel/proc/sysroutine.c' [94.14.28]

VEDERE ANCHE

chdir(2) [87.6].

87.22 os32: getgid(2)

[«](#)

NOME

'*getgid*', '*getegid*' - determinazione del gruppo reale ed efficace

SINTASSI

```
#include <unistd.h>
gid_t getgid (void);
gid_t getegid (void);
```

DESCRIZIONE

La funzione *getgid()* restituisce il numero corrispondente al gruppo reale a cui appartiene il processo; la funzione *getegid()* restituisce il numero del gruppo efficace del processo.

VALORE RESTITUITO

Il numero GID, reale o efficace del processo chiamante. Non sono previsti casi di errore.

DIFETTI

Le funzioni *getgid()* e *getegid()* di os32 devono comunicare con il kernel per ottenere l'informazione che a loro serve, perché la «u-area» (*User area*) è trattenuta all'interno del kernel stesso.

FILE SORGENTI

'lib/unistd.h' [[95.30](#)]

'lib/unistd/getgid.c' [[95.30.22](#)]

'lib/unistd/getegid.c' [[95.30.20](#)]

'lib/sys/os32/sys.s' [[95.21.7](#)]

'kernel/ibm_i386/isr.s' [[94.6.21](#)]

'kernel/proc/sysroutine.c' [[94.14.28](#)]

VEDERE ANCHE

setgid(2) [[87.48](#)], *getuid(2)* [[87.27](#)], *geteuid(2)* [[87.27](#)].

87.23 os32: *geteuid(2)*

« Vedere *getuid(2)* [[87.27](#)].

87.24 os32: *getpgrp(2)*

« Vedere *getpid(2)* [[87.25](#)].

87.25 os32: getpid(2)



NOME

‘**getpid**’, ‘**getppid**’, ‘**getpgrp**’ - determinazione del numero del processo o del gruppo di processi

SINTASSI

```
#include <unistd.h>
pid_t getpid (void);
pid_t getppid (void);
pid_t getpgrp (void);
```

DESCRIZIONE

La funzione *getpid()* restituisce il numero del processo chiamante; la funzione *getppid()* restituisce il numero del processo genitore rispetto a quello chiamante; la funzione *getpgrp()* restituisce il numero attribuito al gruppo di processi a cui appartiene quello chiamante.

VALORE RESTITUITO

Il numero di processo o di gruppo di processi, relativo al contesto della funzione. Non sono previsti casi di errore.

DIFETTI

Le funzioni *getpid()*, *getppid()* e *getpgrp()* di os32 devono comunicare con il kernel per ottenere l’informazione che a loro serve, perché la «u-area» (*User area*) è trattenuta all’interno del kernel stesso.

FILE SORGENTI

‘lib/unistd.h’ [[95.30](#)]

'lib/unistd/getpid.c' [95.30.25]
'lib/unistd/getppid.c' [95.30.26]
'lib/unistd/getpgrp.c' [95.30.24]
'lib/sys/os32/sys.s' [95.21.7]
'kernel/ibm_i386/isr.s' [94.6.21]
'kernel/proc/sysroutine.c' [94.14.28]

VEDERE ANCHE

getuid(2) [87.27] *fork(2)* [87.19], *execve(2)* [87.14].

87.26 os32: getppid(2)

« Vedere *getpid(2)* [87.25].

87.27 os32: getuid(2)

«

NOME

'**getuid**', '**geteuid**' - determinazione dell'identità reale ed efficace

SINTASSI

```
#include <unistd.h>
uid_t getuid (void);
uid_t geteuid (void);
```

DESCRIZIONE

La funzione *getuid()* restituisce il numero corrispondente all'identità reale del processo; la funzione *geteuid()* restituisce il numero dell'identità efficace del processo.

VALORE RESTITUITO

Il numero UID, reale o efficace del processo chiamante. Non sono previsti casi di errore.

DIFETTI

Le funzioni *getuid()* e *geteuid()* di os32 devono comunicare con il kernel per ottenere l'informazione che a loro serve, perché la «u-area» (*User area*) è trattenuta all'interno del kernel stesso.

FILE SORGENTI

'lib/unistd.h' [95.30]

'lib/unistd/getuid.c' [95.30.27]

'lib/unistd/geteuid.c' [95.30.21]

'lib/sys/os32/sys.s' [95.21.7]

'kernel/ibm_i386/isr.s' [94.6.21]

'kernel/proc/sysroutine.c' [94.14.28]

VEDERE ANCHE

setuid(2) [87.51], *getgid(2)* [87.22], *getegid(2)* [87.22].

87.28 os32: ipconfig(2)

NOME

'**ipconfig**' - configurazione di un'interfaccia di rete con l'indirizzo IPv4 e la maschera di rete (funzione specifica di os32)

SINTASSI

```
#include <sys/os32.h>
int ipconfig (int n, in_addr_t address, int m);
```

DESCRIZIONE

La funzione di sistema *ipconfig()*, specifica di os32, permette di configurare un'interfaccia di rete con il suo indirizzo IPv4 e la sua maschera di rete. Il primo parametro, *n*, individua l'interfaccia di rete, nella tabella *net_table[]*; pertanto, lo zero è riservato all'indirizzo locale virtuale (*loopback*), pari all'interfaccia '**net0**', mentre i valori successivi riguardano le interfacce Ethernet reali. L'ultimo parametro, *m*, è la maschera di rete, espressa in quantità di bit; per esempio, il valore 16 corrisponderebbe alla maschera 255.255.0.0.

VALORE RESTITUITO

In caso di successo la funzione restituisce zero, altrimenti, in caso di errore, si ottiene -1 e l'aggiornamento della variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EPERM	È possibile usare la funzione soltanto con UID efficace pari a zero; diversamente si ottiene questo errore.
EINVAL	È stato richiesto un numero di interfaccia oltre i limiti della tabella <i>net_table[]</i> .
ENODEV	È stata richiesta un'interfaccia inesistente.

FILE SORGENTI

'lib/sys/os32.h' [95.21]

'lib/sys/os32/ipconfig.c' [95.21.2]

'kernel/lib_s/s_ipconfig.c' [94.8.18]

VEDERE ANCHE

routeadd(2) [87.42], *routedel(2)* [87.43].

87.29 os32: kill(2)



NOME

‘**kill**’ - invio di un segnale a un processo

SINTASSI

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig)
```

DESCRIZIONE

La funzione *kill()* invia il segnale *sig* al processo numero *pid*, oppure a un gruppo di processi. Questa realizzazione particolare di os32 comporta come segue:

- se il valore *pid* è maggiore di zero, il segnale viene inviato al processo con il numero *pid*, ammesso di averne il permesso;
- se il valore *pid* è pari a zero, il segnale viene inviato a tutti i processi appartenenti allo stesso utente (quelli che hanno la stessa identità efficace, ovvero il valore *eu*id), ma se il processo che chiama la funzione lavora con un valore di *eu*id pari a zero, il segnale viene inviato a tutti i processi, a partire dal numero due (si salta ‘**init**’);
- valori negativi di *pid* non vengono presi in considerazione.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EPERM	Il processo non ha i permessi per inviare il segnale alla destinazione richiesta.
ESRCH	La ricerca del processo <i>pid</i> è fallita. Nel caso di os32, si ottiene questo errore anche per valori negativi di <i>pid</i> .

FILE SORGENTI

'lib/sys/types.h' [95.26]

'lib/signal.h' [95.17]

'lib/signal/kill.c' [95.17.2]

'lib/sys/os32/sys.s' [95.21.7]

'kernel/ibm_i386/isr.s' [94.6.21]

'kernel/proc/sysroutine.c' [94.14.28]

'kernel/lib_s/s_kill.c' [94.8.19]

VEDERE ANCHE

signal(2) [87.52].

87.30 os32: link(2)

«

NOME

'**link**' - creazione di un collegamento fisico tra un file esistente e un altro nome

SINTASSI

```
#include <unistd.h>
int link (const char *path_old, const char *path_new);
```

DESCRIZIONE

La funzione *link()* produce un nuovo collegamento a un file già esistente. Va fornito il percorso del file già esistente, *path_old* e quello del file da creare, in qualità di collegamento, *path_new*. L'operazione può avvenire soltanto se i due percorsi si trovano sulla stessa unità di memorizzazione e se ci sono i permessi di scrittura necessari nella directory di destinazione. Dopo l'operazione di collegamento, fatta in questo modo, non è possibile distinguere quale sia il file originale e quale sia invece il nome aggiunto.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
EPERM	Operazione non consentita.
EEXIST	Il nome da creare esiste già.
EACCES	Accesso non consentito.
ENOENT	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
EROFS	Il file system consente soltanto un accesso in lettura.
ENOTDIR	Uno dei due percorsi non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.

FILE SORGENTI

‘lib/unistd.h’ [[95.30](#)]

‘lib/unistd/link.c’ [[95.30.29](#)]

‘lib/sys/os32/sys.s’ [[95.21.7](#)]

‘kernel/ibm_i386/isr.s’ [[94.6.21](#)]

‘kernel/proc/sysroutine.c’ [[94.14.28](#)]

‘kernel/lib_s/s_link.c’ [[94.8.20](#)]

VEDERE ANCHE

ln(1) [[86.13](#)] *open(2)* [[87.37](#)], *stat(2)* [[87.55](#)], *unlink(2)* [[87.62](#)].

87.31 os32: listen(2)



NOME

‘**listen**’ - in ascolto attendendo una connessione verso un socket locale

SINTASSI

```
#include <sys/socket.h>
int listen (int sfdn, int backlog);
```

DESCRIZIONE

La funzione di sistema *listen()* viene usata per mettere un socket in attesa di connessione dall'esterno. Di norma, prima di usare questa funzione ci si avvale di *bind()* [87.4], per impostare le caratteristiche principali del socket.

Il parametro *backlog* serve a specificare il numero massimo di richieste di connessione che si possono accodare.

Per mettere in atto effettivamente una nuova connessione, partendo dalla prima richiesta disponibile, accodata da *listen()*, si usa poi la funzione *accept()* [87.3].

VALORE RESTITUITO

In caso di successo la funzione restituisce zero; in presenza di errori restituisce invece -1 e aggiorna la variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore <i>sfdn</i> non è valido.
ENOTSOCK	Il descrittore <i>sfdn</i> non è un socket.
EOPNOTSUPP	Il descrittore <i>sfdn</i> non è un socket di tipo SOCK_STREAM .
EISCONN	Il descrittore <i>sfdn</i> corrisponde a un socket già connesso o in un altro stato che non può essere cambiato.
EADDRINUSE	Un altro socket in ascolto sta già utilizzando la stessa porta locale.

FILE SORGENTI

‘lib/sys/socket.h’ [95.23]

‘lib/sys/socket/listen.c’ [95.23.4]

‘kernel/lib_s/s_listen.c’ [94.8.21]

VEDERE ANCHE

bind(2) [87.4], *connect*(2) [87.11], *accept*(2) [87.3], *socket*(2) [87.54].

87.32 os32: longjmp(2)

« Vedere *setjmp*(2) [87.49].

87.33 os32: lseek(2)



NOME

‘**lseek**’ - riposizionamento dell’indice di accesso a un descrittore di file

SINTASSI

```
#include <unistd.h>
off_t lseek (int fdn, off_t offset, int whence);
```

DESCRIZIONE

La funzione *lseek()* consente di riposizionare l’indice di accesso interno al descrittore di file *fdn*. Per fare questo occorre prima determinare un punto di riferimento, rappresentato dal parametro *whence*, dove va usata una macro-variabile definita nel file ‘`unistd.h`’. Può trattarsi dei casi seguenti.

Valore di <i>whence</i>	Significato
SEEK_SET	Lo scostamento si riferisce all’inizio del file.
SEEK_CUR	Lo scostamento si riferisce alla posizione che ha già l’indice interno al file.
SEEK_END	Lo scostamento si riferisce alla fine del file.

Lo scostamento indicato dal parametro *offset* si applica a partire dalla posizione a cui si riferisce *whence*, pertanto può avere segno positivo o negativo, ma in ogni caso non è possibile collocare l’indice prima dell’inizio del file.

VALORE RESTITUITO

Se l'operazione avviene con successo, la funzione restituisce il valore dell'indice riposizionato, preso come scostamento a partire dall'inizio del file. In caso di errore, restituisce invece il valore -1 , aggiornando di conseguenza anche la variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il numero del descrittore di file non è valido.
EINVAL	Il valore di <i>whence</i> non è contemplato, oppure la combinazione tra <i>whence</i> e <i>offset</i> non è valida.

FILE SORGENTI

'lib/unistd.h' [95.30]

'lib/unistd/lseek.c' [95.30.30]

'lib/sys/os32/sys.s' [95.21.7]

'kernel/ibm_i386/isr.s' [94.6.21]

'kernel/proc/sysroutine.c' [94.14.28]

'kernel/lib_s/s_lseek.c' [94.8.23]

VEDERE ANCHE

dup(2) [87.12] *fork(2)* [87.19], *open(2)* [87.37], *fseek(3)* [88.44].

87.34 os32: mkdir(2)

«

NOME

'**mkdir**' - creazione di una directory

SINTASSI

```
#include <sys/stat.h>
int mkdir (const char *path, mode_t mode);
```

DESCRIZIONE

La funzione *mkdir()* crea una directory, indicata attraverso un percorso, nel parametro *path*, specificando la modalità dei permessi, con il parametro *mode*.

Tuttavia, il valore del parametro *mode* non viene preso in considerazione integralmente: di questo si considerano solo gli ultimi nove bit, ovvero quelli dei permessi di utenti, gruppi e altri utenti; inoltre, vengono tolti i bit presenti nella maschera dei permessi associata al processo (si veda anche *umask(2)* [87.60]).

La directory che viene creata in questo modo, appartiene all'identità efficace del processo, ovvero all'utente per conto del quale questo sta funzionando.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il percorso indicato non è valido.
EEXIST	Esiste già un file o una directory con lo stesso nome.
ENOTDIR	Una porzione del percorso della directory da creare, non è una directory.
ENOENT	Una porzione del percorso della directory da creare non esiste.
EACCES	Permesso negato.

FILE SORGENTI

‘lib/sys/stat.h’ [95.25]

‘lib/sys/stat/mkdir.c’ [95.25.4]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

‘kernel/lib_s/s_mkdir.c’ [94.8.24]

VEDERE ANCHE

mkdir(1) [86.17], *chmod(2)* [87.7], *chown(2)* [87.8], *mknod(2)* [87.35], *mount(2)* [87.36], *stat(2)* [87.55], *umask(2)* [87.60], *unlink(2)* [87.62].

87.35 os32: mknod(2)



NOME

‘**mknod**’ - creazione di un file vuoto di qualunque tipo

SINTASSI

```
#include <sys/stat.h>
int mknod (const char *path, mode_t mode, dev_t device);
```

DESCRIZIONE

La funzione *mknod()* crea un file vuoto, di qualunque tipo. Potenzialmente può creare anche una directory, ma priva di qualunque voce, rendendola così non adeguata al suo scopo (una directory richiede almeno le voci ‘.’ e ‘..’, per potersi considerare tale).

Il parametro *path* specifica il percorso del file da creare; il parametro *mode* serve a indicare il tipo di file da creare, oltre ai permessi comuni.

Il parametro *device*, con il quale va indicato il numero di un dispositivo (completo di numero primario e secondario), viene preso in considerazione soltanto se nel parametro *mode* si richiede la creazione di un file di dispositivo a caratteri o a blocchi.

Il valore del parametro *mode* va costruito combinando assieme delle macro-variabili definite nel file ‘sys/stat.h’, come descritto nella pagina di manuale *stat(2)* [87.55], tenendo conto che os32 non può gestire file FIFO, collegamenti simbolici e socket di dominio Unix.

Il valore del parametro *mode*, per la porzione che riguarda i permessi di accesso al file, viene comunque filtrato con la maschera dei permessi (*umask(2)* [87.55]).

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il percorso indicato non è valido.
EEXIST	Esiste già un file o una directory con lo stesso nome.
ENOTDIR	Una porzione del percorso del file da creare, non è una directory.
ENOENT	Una porzione del percorso del file da creare non esiste.
EACCES	Permesso negato.

FILE SORGENTI

‘lib/sys/stat.h’ [95.25]

‘lib/sys/stat/mknod.c’ [95.25.5]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

‘kernel/lib_s/s_mknod.c’ [94.8.25]

VEDERE ANCHE

mkdir(2) [87.34], *chmod(2)* [87.7], *chown(2)* [87.8], *fcntl(2)* [87.18], *stat(2)* [87.55], *umask(2)* [87.60], *unlink(2)* [87.62].

87.36 os32: mount(2)



NOME

‘**mount**’, ‘**umount**’ - innesto e distacco di unità di memorizzazione

SINTASSI

```
#include <sys/os32.h>
int mount (const char *path_dev, const char *path_mnt,
           int options);
int umount (const char *path_mnt);
```

DESCRIZIONE

La funzione *mount()* permette l’innesto di un’unità di memorizzazione individuata attraverso il percorso del file di dispositivo nel parametro *path_dev*, nella directory corrispondente al percorso *path_mnt*, con le opzioni indicate numericamente nell’ultimo argomento *options*. Le opzioni di innesto, rappresentate attraverso delle macro-variabili, sono solo due:

Opzione	Descrizione
MOUNT_DEFAULT	Innesto normale, in lettura e scrittura.
MOUNT_RO	Innesto in sola lettura.

La funzione *umount()* consente di staccare un innesto fatto precedentemente, specificando il percorso della directory in cui questo è avvenuto.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
-1	Errore: va verificato il contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EPERM	Problema di accesso dovuto alla mancanza dei permessi necessari.
ENOTDIR	Ciò che dovrebbe essere una directory, non lo è.
EBUSY	La directory innesta già un file system e non può innestare un altro.
ENOENT	La directory non esiste.
E_NOT_MOUNTED	La directory non innesta un file system da staccare.
EUNKNOWN	Si è verificato un problema non previsto e sconosciuto.

FILE SORGENTI

'lib/sys/os32.h' [95.21]

'lib/sys/os32/mount.c' [95.21.3]

'lib/sys/os32/umount.c' [95.21.8]

'lib/sys/os32/sys.s' [95.21.7]

'kernel/ibm_i386/isr.s' [94.6.21]

'kernel/proc/sysroutine.c' [94.14.28]

‘kernel/lib_s/s_mount.c’ [94.8.26]

‘kernel/lib_s/s_umount.c’ [94.8.47]

VEDERE ANCHE

mount(8) [92.7], *umount(8)* [92.7].

87.37 os32: open(2)



NOME

‘**open**’ - apertura di un file puro e semplice oppure di un file di dispositivo

SINTASSI

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *path, int oflags);
int open (const char *path, int oflags, mode_t mode);
```

DESCRIZIONE

La funzione *open()* apre un file, indicato attraverso il percorso *path*, in base alle opzioni rappresentate dagli indicatori *oflags*. A seconda del tipo di indicatori specificati, potrebbe essere richiesto il parametro *mode*.

Quando la funzione porta a termine correttamente il proprio compito, restituisce il numero del descrittore del file associato, il quale è sempre quello di valore più basso disponibile per il processo elaborativo in corso.

Il descrittore di file ottenuto inizialmente con la funzione *open()*, è legato al processo elaborativo in corso; tuttavia, se successivamente il processo si sdoppia attraverso la funzione *fork()*, tale descrittore, se ancora aperto, viene duplicato nella nuova copia del processo. Inoltre, se per il descrittore aperto non viene impostato l'indicatore 'FD_CLOEXEC' (con l'ausilio della funzione *fcntl()*), se il processo viene rimpiazzato con la funzione *execve()*, il descrittore aperto viene ereditato dal nuovo processo. Il parametro *oflags* richiede necessariamente la specificazione della modalità di accesso, attraverso la combinazione appropriata dei valori: 'O_RDONLY', 'O_WRONLY', 'O_RDWR'. Inoltre, si possono combinare altri indicatori: 'O_CREAT', 'O_TRUNC', 'O_APPEND'.

Opzione	Descrizione
O_RDONLY	Richiede un accesso in lettura.
O_WRONLY	Richiede un accesso in scrittura.
O_RDWR O_RDONLY O_WRONLY	Richiede un accesso in lettura e scrittura (la combinazione di 'O_RDONLY' e di 'O_WRONLY' è equivalente all'uso di 'O_RDWR').
O_CREAT	Richiede di creare contestualmente il file, ma in tal caso va usato anche il parametro <i>mode</i> .
O_TRUNC	Se file da aprire esiste già, richiede che questo sia ridotto preventivamente a un file vuoto.
O_APPEND	Fa in modo che le operazioni di scrittura avvengano sempre partendo dalla fine del file.

Quando si utilizza l'opzione `'O_CREAT'`, è necessario stabilire la modalità dei permessi, cosa che va fatta preferibilmente attraverso la combinazione di costanti simboliche appropriate, come elencato nella tabella successiva. Tale combinazione va fatta con l'uso dell'operatore OR binario; per esempio: `'S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH'`. Va osservato che os32 non gestisce i gruppi di utenti, pertanto, la definizione dei permessi relativi agli utenti appartenenti al gruppo proprietario di un file, non ha poi effetti pratici nel controllo degli accessi per tale tipo di contesto.

Modalità simbolica	Modalità numerica	Descrizione
<code>S_IRWXU</code>	<code>00700₈</code>	Lettura, scrittura ed esecuzione per l'utente proprietario.
<code>S_IRUSR</code>	<code>00400₈</code>	Lettura per l'utente proprietario.
<code>S_IWUSR</code>	<code>00200₈</code>	Scrittura per l'utente proprietario.
<code>S_IXUSR</code>	<code>00100₈</code>	Esecuzione per l'utente proprietario.

Modalità simbolica	Modalità numerica	Descrizione
<code>S_IRWXG</code>	<code>00070₈</code>	Lettura, scrittura ed esecuzione per il gruppo.
<code>S_IRGRP</code>	<code>00040₈</code>	Lettura per il gruppo.
<code>S_IWGRP</code>	<code>00020₈</code>	Scrittura per il gruppo.
<code>S_IXGRP</code>	<code>00010₈</code>	Esecuzione per il gruppo.

Modalità simbolica	Modalità numerica	Descrizione
S_IRWXO	00007 ₈	Lettura, scrittura ed esecuzione per gli altri utenti.
S_IROTH	00004 ₈	Lettura per gli altri utenti.
S_IWOTH	00002 ₈	Scrittura per gli altri utenti.
S_IXOTH	00001 ₈	Esecuzione per gli altri utenti.

VALORE RESTITUITO

La funzione restituisce il numero del descrittore del file aperto, se l'operazione ha avuto successo, altrimenti dà semplicemente -1 , impostando di conseguenza il valore della variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
EPERM	Operazione non consentita.
EEXIST	Il file da creare esiste già.
EACCES	Accesso non consentito.
ENOENT	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
EROFS	Avendo richiesto un accesso in scrittura, si ottiene che il file system che lo contiene consente soltanto un accesso in lettura.
ENOTDIR	Il percorso che porta al file da aprire non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.
ENFILE	Non si possono aprire altri file nell'ambito del sistema operativo (il sistema ha raggiunto il limite).
EMFILE	Non si possono aprire altri file nell'ambito del processo in corso.

FILE SORGENTI

'lib/sys/types.h' [95.26]

'lib/sys/stat.h' [95.25]

'lib/fcntl.h' [95.6]

'lib/fcntl/open.c' [95.6.3]

VEDERE ANCHE

chmod(2) [87.7], *chown(2)* [87.8], *close(2)* [87.10], *dup(2)* [87.12], *fcntl(2)* [87.18], *link(2)* [87.33], *mknod(2)* [87.35],

mount(2) [87.36], *read(2)* [87.39], *stat(2)* [87.55], *umask(2)* [87.60], *unlink(2)* [87.62], *write(2)* [87.64], *fopen(3)* [88.36].

87.38 os32: pipe(2)

<<

NOME

‘**pipe**’ - creazione di un condotto senza nome

SINTASSI

```
#include <unistd.h>
int pipe (int pipefd[2]);
```

DESCRIZIONE

La funzione *pipe()* crea, nella tabella degli inode, un condotto, ovvero un inode speciale con questa caratteristica. All’inode associa due descrittori, uno per la lettura e l’altro per la scrittura, restituendone il numero, rispettivamente in *pipefd[0]* e *pipefd[1]*. Ciò che viene scritto attraverso il descrittore *pipefd[1]* viene accumulato in una memoria tampone (costituita dallo spazio di memoria inutilizzato nell’inode che lo rappresenta) e viene prelevato con la lettura dal descrittore *pipefd[0]*.

VALORE RESTITUITO

La funzione restituisce zero se la creazione si è conclusa con successo, oppure -1 in caso di problemi, aggiornando di conseguenza il valore di *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	L'argomento fornito non è valido.
EMFILE	Troppi file aperti.
ENFILE	Troppi file aperti nel sistema.

FILE SORGENTI

'lib/unistd.h' [95.30]

'lib/unistd/pipe.c' [95.30.31]

'lib/sys/os32/sys.s' [95.21.7]

'kernel/lib_s/s_pipe.c' [94.8.28]

VEDERE ANCHE

close(2) [87.10] *open(2)* [87.37], *write(2)* [87.64], *read(2)* [87.39].

87.39 os32: read(2)



NOME

'**read**' - lettura di descrittore di file

SINTASSI

```
#include <unistd.h>
ssize_t read (int fdn, void *buffer, size_t count);
```

DESCRIZIONE

La funzione *read()* cerca di leggere il file rappresentato dal descrittore *fdn*, partendo dalla posizione in cui si trova l'indice interno di accesso, per un massimo di *count* byte, collocando i dati letti in memoria a partire dal puntatore *buffer*. L'indice interno al file viene fatto avanzare della quantità di byte letti effettivamente, se invece si incontra la fine del file, viene aggiornato l'indicatore interno per segnalare tale fatto.

VALORE RESTITUITO

La funzione restituisce la quantità di byte letti effettivamente, oppure zero se è stata raggiunta la fine del file e non si può proseguire oltre. Va osservato che la lettura effettiva di una quantità inferiore di byte rispetto a quanto richiesto non costituisce un errore: in quel caso i byte mancanti vanno richiesti con successive operazioni di lettura. In caso di errore, la funzione restituisce il valore -1 , aggiornando contestualmente la variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il numero del descrittore di file non è valido.
EINVAL	Il file non è aperto in lettura.
E_FILE_TYPE_UNSUPPORTED	Il file è di tipo non gestibile con os32.

FILE SORGENTI

'lib/unistd.h' [95.30]

'lib/unistd/read.c' [95.30.32]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/lib_s/s_read.c’ [94.8.29]

VEDERE ANCHE

close(2) [87.10] *open(2)* [87.37], *write(2)* [87.64].

87.40 os32: recvfrom(2)



NOME

‘**recvfrom**’ - ricezione di un messaggio da un socket

SINTASSI

```
#include <sys/socket.h>
ssize_t recvfrom (int sfdn, void *buffer, size_t count,
                 int flags, struct sockaddr *addrfrom,
                 socklen_t *addrlen);
```

DESCRIZIONE

La funzione *recvfrom()* consente di ricevere un «messaggio» da un socket, collocandolo in memoria a partire dall’indirizzo *buffer*, utilizzando al massimo *count* byte. La funzione restituisce poi la quantità di byte letti, o un valore negativo in caso di errore.

Se la funzione riceve un puntatore valido in corrispondenza di *addrfrom* e di *addrlen*, significa che si vuole annotare in corrispondenza di **addrfrom* l’indirizzo di origine del messaggio ricevuto, in forma di variabile strutturata di tipo ‘**struct sockaddr**’. In tal caso, alla chiamata della funzione il valore di **addrlen* deve indicare la dimensione massima disponibile in

memoria per annotare tale informazione, sapendo che questo valore viene poi modificato per contenere la dimensione originale effettiva: in pratica, se questa dimensione è maggiore di quella della chiamata, vuol dire che tale informazione è stata annotata ma solo in parzialmente, troncandola.

Nella realizzazione di `os32`, il parametro *flags* viene semplicemente ignorato, non essendo previsti indicatori che possano modificare la modalità di ricezione-lettura dei dati.

È importante osservare che la ricezione di un messaggio può risultare troncata, se la memoria tampone che parte da *buffer* non ha una dimensione sufficiente. Questo succede per esempio se si riceve da un socket relativo a una connessione UDP. Quando invece si sta operando con un socket TCP, il flusso di ricezione avviene in modo continuo, senza troncamenti.

La lettura avviene normalmente bloccando il processo chiamante, fino a che si ottiene qualcosa. Diversamente, con l'ausilio della funzione *fcntl()* è possibile attribuire l'opzione *O_NONBLOCK* per lasciare che la funzione termini ugualmente segnalando l'errore *EAGAIN*.

VALORE RESTITUITO

In caso di successo la funzione restituisce la dimensione del messaggio ricevuto; altrimenti si ottiene `-1` e l'aggiornamento della variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore <i>sfdn</i> non è valido.
ENOTSOCK	Il descrittore <i>sfdn</i> non è un socket.
EINVAL	Il puntatore <i>buffer</i> non è valido.
EPROTONOSUPPORT	Il protocollo del socket non è gestito da os32 in questo contesto.
EAFNOSUPPORT	Il tipo di indirizzamento del socket <i>sfdn</i> è diverso da <i>AF_INET</i> .
EAGAIN	Non è disponibile alcun messaggio per il momento.

FILE SORGENTI

‘lib/sys/socket.h’ [95.23]

‘lib/sys/socket/recvfrom.c’ [95.23.5]

‘kernel/lib_s/s_recvfrom.c’ [94.8.30]

VEDERE ANCHE

accept(2) [87.3], *bind(2)* [87.4], *connect(2)* [87.11], *listen(2)* [87.31], *socket(2)* [87.54].

87.41 os32: rmdir(2)



NOME

‘**rmdir**’ - eliminazione di una directory vuota

SINTASSI

```
#include <unistd.h>
int rmdir (const char *path);
```

DESCRIZIONE

La funzione *rmdir()* cancella la directory indicata come percorso, nella stringa *path*, purché sia vuota.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il percorso <i>path</i> non è valido o è semplicemente un puntatore nullo.
ENOTDIR	Il nome indicato o le posizioni intermedie del percorso si riferiscono a qualcosa che non è una directory.
ENOTEMPTY	La directory che si vorrebbe cancellare non è vuota.
EROFS	La directory si trova in un'unità innestata in sola lettura.
EPERM	Mancano i permessi necessari per eseguire l'operazione.
EUNKNOWN	Si è verificato un errore imprevisto e sconosciuto.

FILE SORGENTI

'lib/unistd.h' [95.30]

`'lib/unistd/rmdir.c'` [[95.30.33](#)]

`'lib/sys/os32/sys.s'` [[95.21.7](#)]

`'kernel/lib_s/s_unlink.c'` [[94.8.48](#)]

VEDERE ANCHE

`mkdir(2)` [[87.34](#)], `unlink(2)` [[87.62](#)].

87.42 os32: routeadd(2)



NOME

`'routeadd'` - aggiunta di un instradamento nella tabella degli instradamenti (funzione specifica di os32)

SINTASSI

```
#include <sys/os32.h>
int routeadd (in_addr_t destination, int m,
              in_addr_t router, int device);
```

DESCRIZIONE

La funzione di sistema `routeadd()`, specifica di os32, permette di aggiungere un instradamento IPv4 che richiede l'attraversamento di un router. Infatti, gli instradamenti nella rete locale sono definiti automaticamente dalla funzione `ipconfig()` [[87.28](#)], contestualmente alla configurazione dell'interfaccia.

I parametri della funzione sono rappresentati rispettivamente da: indirizzo di destinazione; maschera in forma di quantità di bit; indirizzo del router da interpellare; numero dell'interfaccia di rete locale, attraverso la quale eseguire la comunicazione.

VALORE RESTITUITO

In caso di successo la funzione restituisce zero, altrimenti, in caso di errore, si ottiene -1 e l'aggiornamento della variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EPERM	È possibile usare la funzione soltanto con UID efficace pari a zero; diversamente si ottiene questo errore.
EINVAL	È stata indicata un'interfaccia di rete impossibile (sono ammissibili solo valori da 0 a 32).
ENOMEM	Non c'è spazio per aggiungere l'instradamento nella tabella relativa.

FILE SORGENTI

'lib/sys/os32.h' [[95.21](#)]

'lib/sys/os32/routeadd.c' [[95.21.5](#)]

'kernel/lib_s/s_routeadd.c' [[94.8.31](#)]

VEDERE ANCHE

ipconfig(2) [[87.28](#)], *routedel(2)* [[87.43](#)].

87.43 os32: routedel(2)

«

NOME

'**routedel**' - eliminazione di un instradamento nella tabella degli instradamenti (funzione specifica di os32)

SINTASSI

```
#include <sys/os32.h>
int routedel (in_addr_t destination, int m);
```

DESCRIZIONE

La funzione di sistema *routedel()*, specifica di os32, permette di eliminare un instradamento IPv4, individuato dall'indirizzo e dalla maschera di rete (espressa in quantità di bit).

VALORE RESTITUITO

In caso di successo la funzione restituisce zero, altrimenti, in caso di errore, si ottiene -1 e l'aggiornamento della variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EPERM	È possibile usare la funzione soltanto con UID efficace pari a zero; diversamente si ottiene questo errore.
EINVAL	È stata indicata un'interfaccia di rete impossibile (sono ammissibili solo valori da 0 a 32), oppure l'instradamento da cancellare non esiste.

FILE SORGENTI

'lib/sys/os32.h' [[95.21](#)]

'lib/sys/os32/routedel.c' [[95.21.6](#)]

'kernel/lib_s/s_routedel.c' [[94.8.32](#)]

VEDERE ANCHE

ipconfig(2) [[87.28](#)], *routeadd(2)* [[87.42](#)].

87.44 os32: sbrk(2)

«
Vedere *brk(2)* [87.5].

87.45 os32: send(2)

«

NOME

‘**send**’ - invio di un messaggio attraverso un socket

SINTASSI

```
#include <sys/socket.h>
ssize_t send (int sfdn, const void *buffer, size_t count,
              int flags);
```

DESCRIZIONE

La funzione *recvfrom()* consente di inviare un «messaggio» attraverso un socket, prelevandolo dalla memoria a partire dall'indirizzo *buffer*, utilizzando da lì al massimo *count* byte. La funzione restituisce poi la quantità di byte trasmessi effettivamente, o un valore negativo in caso di errore.

Nella realizzazione di os32, il parametro *flags* viene semplicemente ignorato, non essendo previsti indicatori che possano modificare la modalità di ricezione-lettura dei dati. D'altra parte, se non si considera il parametro *flags*, la funzione si comporta nello stesso modo di *write()write(2)* [87.64].

La trasmissione avviene normalmente bloccando il processo chiamante, fino a che si ottiene il risultato. Diversamente, con l'ausilio della funzione *fcntl()fcntl(2)* [87.18] è possibile attribuire l'opzione *O_NONBLOCK* per lasciare che la funzione termini ugualmente segnalando l'errore *EAGAIN*.

VALORE RESTITUITO

In caso di successo la funzione restituisce la dimensione del messaggio trasmesso effettivamente; altrimenti si ottiene -1 e l'aggiornamento della variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore <i>sfdn</i> non è valido.
ENOTSOCK	Il descrittore <i>sfdn</i> non è un socket.
EINVAL	Il puntatore <i>buffer</i> non è valido.
ECONNREFUSED	Non è possibile contattare la controparte alla porta desiderata.
ENOPROTOPT	Protocollo non disponibile.
EHOSTUNREACH	Non è possibile contattare la controparte all'indirizzo desiderato.
ENETUNREACH	Rete irraggiungibile.
EDESTADDRREQ	Indirizzo di destinazione mancante.
EPROTONOSUPPORT	Il protocollo del socket non è gestito da os32 in questo contesto.
EPIPE	La trasmissione in un flusso TCP si è interrotta prematuramente.
EAGAIN	Temporaneamente non è possibile trasmettere.
EAFNOSUPPORT	Il tipo di indirizzamento non è <i>AF_INET</i> , l'unico attualmente ammissibile per os32.

FILE SORGENTI

‘lib/sys/socket.h’ [95.23]

‘lib/sys/socket/send.c’ [95.23.6]

‘kernel/lib_s/s_send.c’ [94.8.34]

VEDERE ANCHE

accept(2) [87.3], *bind(2)* [87.4], *connect(2)* [87.11], *listen(2)* [87.31], *socket(2)* [87.54], *recvfrom(2)* [87.40], *write(2)* [87.64].

87.46 os32: setegid(2)

« Vedere *setgid(2)* [87.48].

87.47 os32: seteuid(2)

« Vedere *setuid(2)* [87.51].

87.48 os32: setgid(2)

«

NOME

‘**setgid**’, ‘**setegid**’ - impostazione dell’identità del gruppo

SINTASSI

```
#include <unistd.h>
int setgid (gid_t gid);
int setegid (gid_t gid);
```

DESCRIZIONE

Ogni processo viene associato a un gruppo di utenti, rappresentato da un numero, noto come GID, ovvero *group identity*. Tuttavia

si distinguono tre tipi di numeri GID: l'identità reale, l'identità efficace e un'identità salvata in precedenza. L'identità efficace di gruppo (EGID) è quella con cui opera sostanzialmente il processo; l'identità salvata è quella che ha avuto il processo in un altro momento in qualità di identità efficace e che per qualche motivo non ha più.

La funzione *setgid()* riceve come argomento un numero GID e si comporta diversamente a seconda della personalità del processo, come descritto nell'elenco successivo:

- se l'identità efficace del processo, EUID o EGID, corrisponde a zero, trattandosi di un caso particolarmente privilegiato, tutte le identità di gruppo del processo (reale, efficace e salvata) vengono inizializzate con il valore fornito alla funzione *setgid()*;
- se l'identità efficace di gruppo del processo corrisponde a quella fornita come argomento a *setgid()*, nulla cambia nella gestione delle identità del processo;
- se l'identità di gruppo reale o quella salvata in precedenza corrispondono a quella fornita come argomento di *setgid()*, viene aggiornato il valore dell'identità efficace, senza cambiare le altre;
- in tutti gli altri casi, l'operazione non è consentita e si ottiene un errore.

La funzione *setegid()* riceve come argomento un numero GID e imposta con tale valore l'identità di gruppo efficace del processo, purché si verifichi almeno una delle condizioni seguenti:

- l'identità EUID o EGID del processo è zero;

- l'identità di gruppo reale o quella salvata del processo corrisponde all'identità efficace che si vuole impostare;
- l'identità di gruppo efficace del processo corrisponde già all'identità efficace che si vuole impostare.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EPERM	Non si dispone dei permessi necessari a eseguire il cambiamento di identità.

FILE SORGENTI

'lib/unistd.h' [95.30]
 'lib/unistd/setgid.c' [95.30.37]
 'lib/unistd/setegid.c' [95.30.35]
 'lib/sys/os32/sys.s' [95.21.7]
 'kernel/ibm_i386/isr.s' [94.6.21]
 'kernel/proc/sysroutine.c' [94.14.28]
 'kernel/lib_s/setgid.c' [94.8.37]
 'kernel/lib_s/setegid.c' [94.8.35]

VEDERE ANCHE

getuid(2) [87.27], *geteuid(2)* [87.27], *getgid(2)* [87.22],
getegid(2) [87.22], *setuid(2)* [87.51].

87.49 os32: setjmp(2)



NOME

‘**set jmp**’, ‘**long jmp**’ - salvataggio e recupero della pila per i «salti non locali»

SINTASSI

```
#include <set jmp.h>
int set jmp ( jmp_buf env );
void long jmp ( jmp_buf env, int val );
```

DESCRIZIONE

La funzione *set jmp()* consente di salvare, in corrispondenza di una posizione di memoria rappresentata da *env*, il contesto della pila dei dati per poterne recuperare lo stato in un momento successivo, attraverso *long jmp()*. Quando la funzione *long jmp()* viene chiamata, la sua uscita si manifesta al posto della chiamata di *set jmp()* a cui è stato fatto riferimento con *env*. Pertanto, quando la funzione *set jmp()* viene chiamata realmente, restituisce sempre il valore zero, mentre quando l’uscita di *set jmp()* deve in realtà manifestare il salto ottenuto con *long jmp()*, il valore restituito è quanto corrisponde a *val* (il secondo parametro di *long jmp()*).

os32 realizza il meccanismo del salto con ripristino del contesto, attraverso chiamate di funzione, per facilitare la comprensibilità del codice.

Figura 87.52. Lo stato della pila durante le varie fasi che riguardano la chiamata di `setjmp()`, a confronto con i tipi `'jmp_stack_t'` e `'jmp_env_t'`.

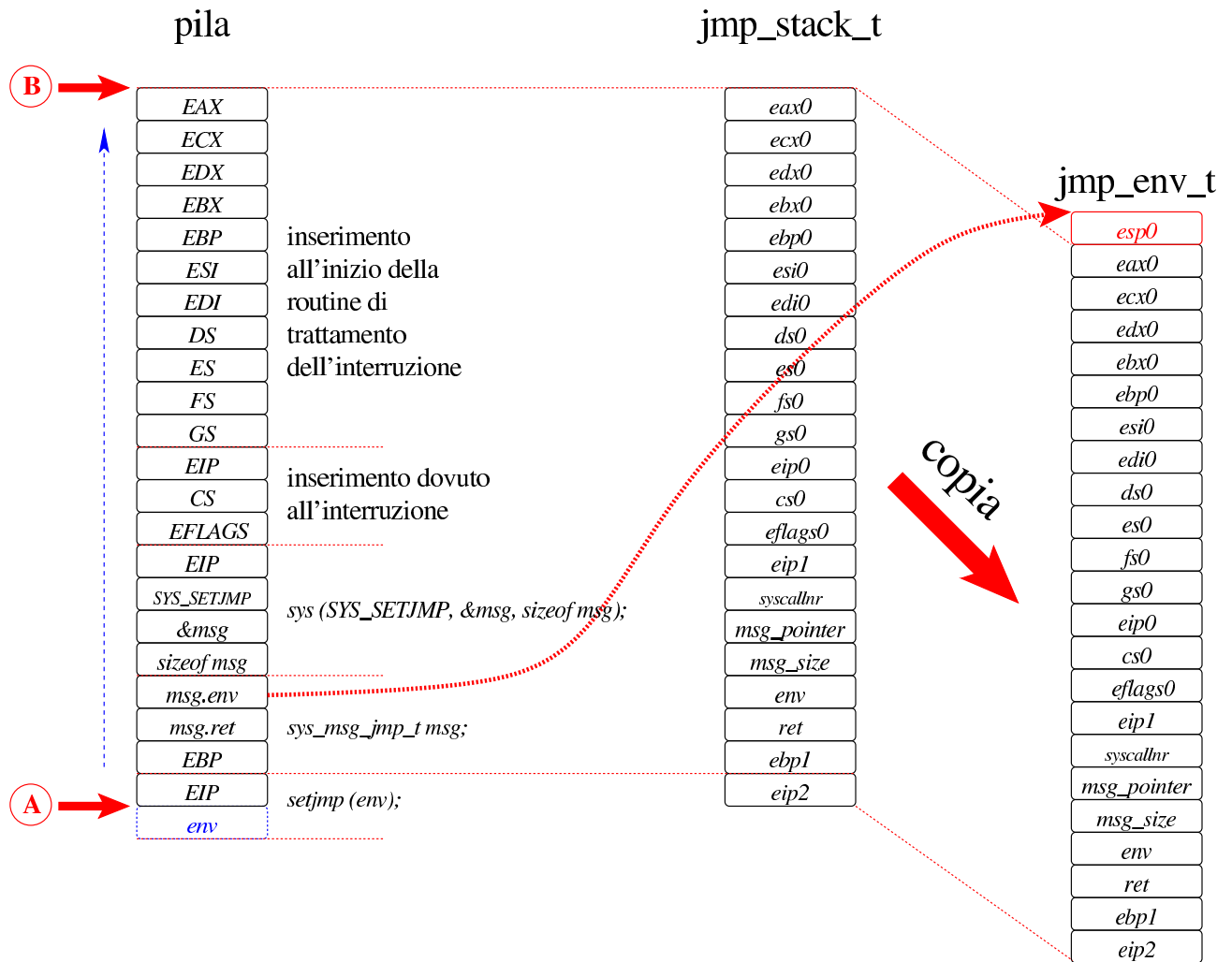
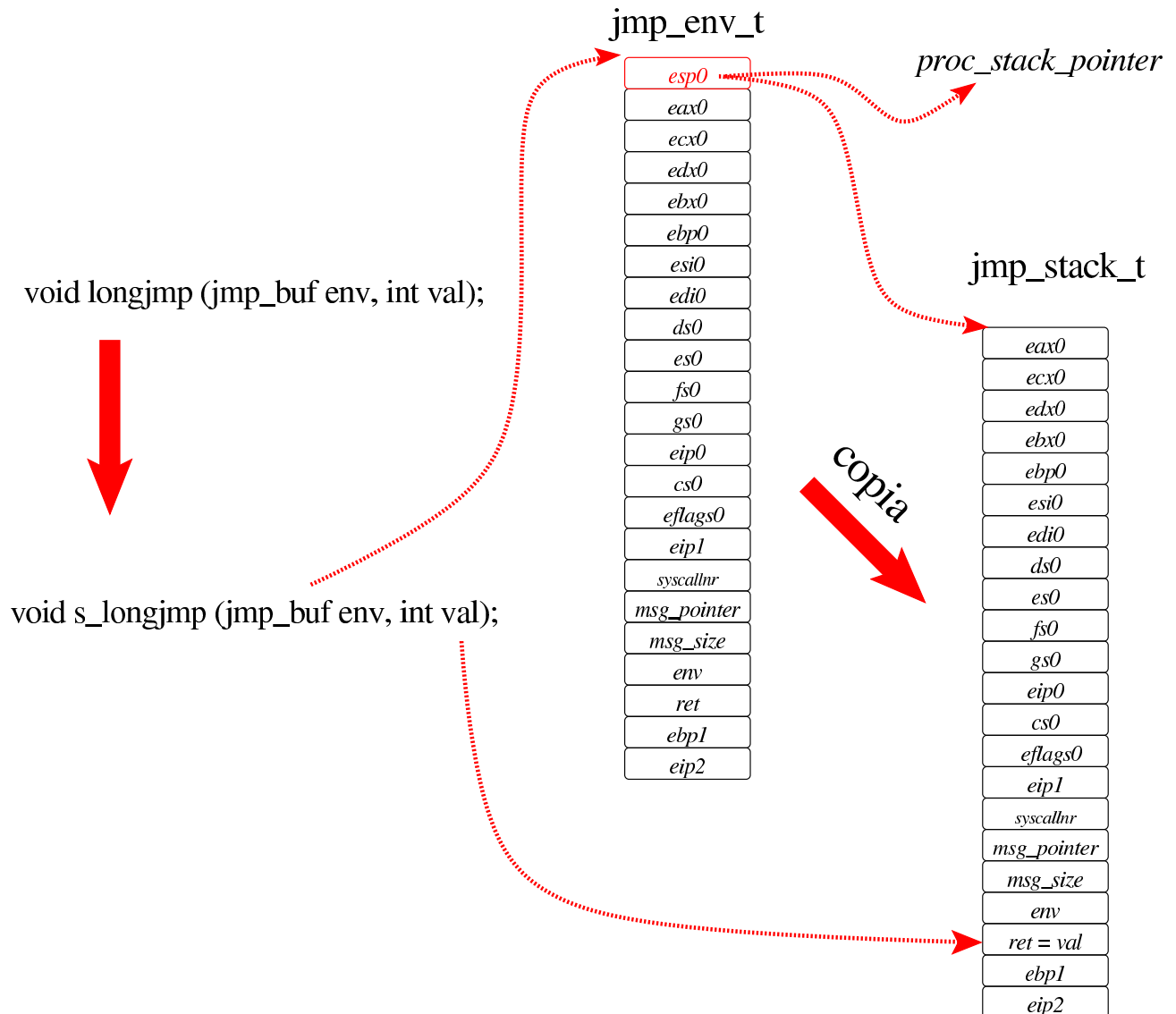


Figura 87.53. La chiamata di *longjmp()* ricostruisce la vecchia pila di *setjmp()*, nella posizione in cui si trovava, ricollocando l'indice della pila e modificando il valore che poi *setjmp()* rediviva va a restituire.



VALORE RESTITUITO

La funzione *setjmp()* restituisce zero quando viene chiamata per salvare il contesto operativo, mentre restituisce *val* quando rappresenta il ripristino del contesto derivante dall'uso della funzione *longjmp()*.

NOTE

Le funzioni *setjmp()* e *longjmp()* fanno parte dello standard per motivi storici, ma il loro uso è decisamente sconsigliabile.

Quando si ripristina il contesto con *longjmp()*, è necessario che la pila precedente alla chiamata di *setjmp()* non sia stata compromessa. Pertanto, come condizione necessaria, ma non sufficiente, *longjmp()* può essere usato soltanto per ripristinare un contesto che nella pila attuale si trovi in una posizione antecedente (più interna).

FILE SORGENTI

‘lib/setjmp.h’ [95.16]

‘lib/setjmp/setjmp.s’ [95.16.2]

‘lib/setjmp/longjmp.c’ [95.16.1]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

‘kernel/lib_s/s_setjmp.c’ [94.8.38]

‘kernel/lib_s/s_longjmp.c’ [94.8.22]

VEDERE ANCHE

signal(2) [87.52].

87.50 os32: setpgrp(2)

«

NOME

‘**setpgrp**’ - impostazione del gruppo a cui appartiene il processo

SINTASSI

```
#include <unistd.h>
int setpgrp (void);
```

DESCRIZIONE

La funzione *setpgrp()* fa sì che il processo in corso costituisca un proprio gruppo autonomo, corrispondente al proprio numero PID. In altri termini, la funzione serve per iniziare un nuovo gruppo di processi, a cui i processi figli creati successivamente vengano associati in modo predefinito.

VALORE RESTITUITO

La funzione termina sempre con successo e restituisce sempre zero.

FILE SORGENTI

‘lib/unistd.h’ [[95.30](#)]

‘lib/unistd/setpgrp.c’ [[95.30.38](#)]

‘lib/sys/os32/sys.s’ [[95.21.7](#)]

‘kernel/ibm_i386/isr.s’ [[94.6.21](#)]

‘kernel/proc/sysroutine.c’ [[94.14.28](#)]

VEDERE ANCHE

getpgrp(2) [[87.50](#)], *getuid(2)* [[87.27](#)].

87.51 os32: setuid(2)

**NOME**

'setuid', **'seteuid'** - impostazione dell'identità dell'utente

SINTASSI

```
#include <unistd.h>
int setuid (uid_t uid);
int seteuid (uid_t uid);
```

DESCRIZIONE

A ogni processo viene attribuita l'identità di un utente, rappresentata da un numero, noto come UID, ovvero *user identity*. Tuttavia si distinguono tre tipi di numeri UID: l'identità reale, l'identità efficace e un'identità salvata in precedenza. L'identità efficace (EUID) è quella con cui opera sostanzialmente il processo; l'identità salvata è quella che ha avuto il processo in un altro momento in qualità di identità efficace e che per qualche motivo non ha più.

La funzione *setuid()* riceve come argomento un numero UID e si comporta diversamente a seconda della personalità del processo, come descritto nell'elenco successivo:

- se l'identità efficace del processo corrisponde a zero, trattandosi di un caso particolarmente privilegiato, tutte le identità del processo (reale, efficace e salvata) vengono inizializzate con il valore fornito alla funzione *setuid()*;

- se l'identità efficace del processo corrisponde a quella fornita come argomento a *setuid()*, nulla cambia nella gestione delle identità del processo;
- se l'identità reale o quella salvata in precedenza corrispondono a quella fornita come argomento di *setuid()*, viene aggiornato il valore dell'identità efficace, senza cambiare le altre;
- in tutti gli altri casi, l'operazione non è consentita e si ottiene un errore.

La funzione *seteuid()* riceve come argomento un numero UID e imposta con tale valore l'identità efficace del processo, purché si verifichi almeno una delle condizioni seguenti:

- l'identità efficace del processo è zero;
- l'identità reale o quella salvata del processo corrisponde all'identità efficace che si vuole impostare;
- l'identità efficace del processo corrisponde già all'identità efficace che si vuole impostare.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EPERM	Non si dispone dei permessi necessari a eseguire il cambiamento di identità.

FILE SORGENTI

'lib/unistd.h' [[95.30](#)]

'lib/unistd/setuid.c' [95.30.39]
'lib/unistd/seteuid.c' [95.30.36]
'lib/sys/os32/sys.s' [95.21.7]
'kernel/ibm_i386/isr.s' [94.6.21]
'kernel/proc/sysroutine.c' [94.14.28]
'kernel/lib_s/s_setuid.c' [94.8.39]
'kernel/lib_s/s_seteuid.c' [94.8.36]

VEDERE ANCHE

getuid(2) [87.27], *geteuid(2)* [87.27], *getgid(2)* [87.22],
getegid(2) [87.22], *setgid(2)* [87.48], *setegid(2)* [87.48].

VEDERE ANCHE

getuid(2) [87.51].

87.52 os32: signal(2)

«

NOME

'**signal**' - abilitazione e disabilitazione dei segnali

SINTASSI

```
#include <signal.h>
sighandler_t signal (int sig, sighandler_t handler);
```

DESCRIZIONE

La funzione *signal()* di os32 consente soltanto di abilitare o disabilitare un segnale, il quale, se abilitato, può essere gestito solo in modo predefinito dal sistema. Pertanto, il valore che

può assumere *handler* sono solo: ‘**SIG_DFL**’ (gestione predefinita) e ‘**SIG_IGN**’ (ignora il segnale). Il primo parametro, *sig*, rappresenta il segnale a cui applicare *handler*.

Il tipo ‘**sighandler_t**’ è definito nel file ‘`signal.h`’, nel modo seguente:

```
typedef void (*sighandler_t) (int);
```

Rappresenta il puntatore a una funzione avente un solo parametro di tipo ‘**int**’, la quale non restituisce alcunché.

VALORE RESTITUITO

La funzione restituisce il tipo di «gestione» impostata precedentemente per il segnale richiesto, ovvero ‘**SIG_ERR**’ in caso di errore.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il numero <i>sig</i> o il valore di <i>handler</i> non sono validi.

NOTE

Lo scopo della funzione *signal()* dovrebbe essere quello di consentire l’associazione di un evento, manifestato da un segnale, all’esecuzione di un’altra funzione, avente una forma del tipo ‘*nome* (**int**)’. `os32` non consente tale gestione, pertanto lascia soltanto la possibilità di attribuire un comportamento predefinito al segnale scelto, oppure di disabilitarlo, ammesso che ciò sia consentito. Sotto questo aspetto, il fatto di dover gestire i valori ‘**SIG_ERR**’, ‘**SIG_DFL**’ e ‘**SIG_IGN**’, come se fossero puntatori

a una funzione, diventa superfluo, ma rimane utile soltanto per mantenere un minimo di conformità con quello che è lo standard della funzione *signal()*.

FILE SORGENTI

'lib/signal.h' [95.17]

'lib/signal/signal.c' [95.17.3]

'kernel/ibm_i386/isr.s' [94.6.21]

'kernel/proc/sysroutine.c' [94.14.28]

'kernel/lib_s/s_signal.c' [94.8.40]

VEDERE ANCHE

kill(2) [87.29].

87.53 os32: sleep(2)

«

NOME

'**sleep**' - pausa volontaria del processo chiamante

SINTASSI

```
#include <unistd.h>
unsigned int sleep (unsigned int seconds);
```

DESCRIZIONE

La funzione *sleep()* chiede di mettere a riposo il processo chiamante per la quantità di secondi indicata come argomento. Il processo può però essere risvegliato prima della conclusione di tale durata, ma in tal caso la funzione restituisce la quantità di secondi che non sono stati usati per il «riposo» del processo.

VALORE RESTITUITO

La funzione restituisce zero se la pausa richiesta è trascorsa completamente; altrimenti, restituisce quanti secondi mancano ancora per completare il tempo di riposo chiesto originariamente. Non si prevede il manifestarsi di errori.

FILE SORGENTI

‘lib/unistd.h’ [95.30]

‘lib/unistd/sleep.c’ [95.30.40]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

VEDERE ANCHE

signal(2) [87.52].

87.54 os32: socket(2)



NOME

‘**socket**’ - crea un socket, definendo solo il tipo e il protocollo

SINTASSI

```
#include <sys/socket.h>
int socket (int family, int type, int protocol);
```

DESCRIZIONE

La funzione *socket()* crea un socket, ovvero un terminale di una comunicazione, restituendone il descrittore numerico, per poi potervi fare riferimento.

Il primo parametro della funzione (*family*)), noto anche come «dominio» del socket, può essere soltanto *AF_INET* per os32, corrispondente a un socket di dominio Internet IPv4; pertanto non è possibile creare socket di dominio Unix.

Il secondo parametro, *type*, definisce la modalità con cui avviene la comunicazione attraverso il socket. Per os32 questa può essere:

Tipo	Significato
SOCK_RAW	Definisce una comunicazione generica che per os32 riguarda soltanto i protocolli ICMP.
SOCK_DGRAM	Definisce una comunicazione a pacchetti indipendenti, di una certa dimensione massima, senza controlli. Per os32, questo tipo di modalità riguarda esclusivamente il protocollo UDP.
SOCK_STREAM	Definisce un flusso di byte ordinato, affidabile, a due vie (trasmissione e ricezione indipendenti). Per os32, questo tipo di modalità riguarda esclusivamente il protocollo TCP.

Il terzo parametro, *protocol* definisce il protocollo di comunicazione. Per os32 può essere:

Protocollo	Descrizione
IPPROTO_ICMP	Protocollo ICMP, da abbinare necessariamente a un tipo <i>SOCK_RAW</i> .
IPPROTO_UDP	Protocollo UDP, da abbinare necessariamente a un tipo <i>SOCK_DGRAM</i> .
IPPROTO_TCP	Protocollo TCP, da abbinare necessariamente a un tipo <i>SOCK_STREAM</i> .

VALORE RESTITUITO

In caso di successo la funzione restituisce il descrittore del socket creato; altrimenti si ottiene `-1` e l'aggiornamento della variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EACCES	È stato richiesto di creare un socket di tipo <i>SOCK_RAW</i> senza i privilegi necessari (serve un UID efficace pari a zero).
EAFNOSUPPORT	Il tipo di indirizzamento non è <i>AF_INET</i> , l'unico attualmente ammissibile per os32.
EPROTONOSUPPORT	Il protocollo del socket non è gestito da os32 in questo contesto.
ENFILE	Troppi file aperti nel sistema.
EMFILE	Troppi file aperti.

FILE SORGENTI

'lib/sys/socket.h' [95.23]

'lib/sys/socket/socket.c' [95.23.7]

'kernel/lib_s/s_socket.c' [94.8.41]

VEDERE ANCHE

socket(7) [91.2], *accept(2)* [87.3], *bind(2)* [87.4], *connect(2)* [87.11], *listen(2)* [87.31].

87.55 os32: stat(2)

**NOME**

‘**stat**’, ‘**fstat**’ - interrogazione dello stato di un file

SINTASSI

```
#include <sys/stat.h>
int stat (const char *path, struct stat *buffer);
int fstat (int fdn, struct stat *buffer);
```

DESCRIZIONE

Le funzioni *stat()* e *fstat()* interrogano il sistema su di un file, per ottenerne le caratteristiche in forma di variabile strutturata di tipo ‘**struct stat**’.

La funzione *stat()* individua il file attraverso una stringa contenente il suo percorso (*path*); la funzione ‘**fstat**’ si riferisce a un file aperto di cui si conosce il numero del descrittore (*fdn*). In entrambi i casi, la struttura che deve accogliere l’esito dell’interrogazione, viene indicata attraverso un puntatore, come ultimo argomento (*buffer*).

La struttura ‘**struct stat**’ è definita nel file ‘`sys/stat.h`’ nel modo seguente:

```
struct stat {
    dev_t      st_dev;      // Device containing the file.
    ino_t      st_ino;     // File serial number (inode
                          // number).
    mode_t     st_mode;    // File type and permissions.
    nlink_t    st_nlink;   // Links to the file.
    uid_t      st_uid;     // Owner user id.
    gid_t      st_gid;     // Owner group id.
    dev_t      st_rdev;    // Device number if it is a device
                          // file.
    off_t      st_size;    // File size.
    time_t     st_atime;   // Last access time.
    time_t     st_mtime;   // Last modification time.
    time_t     st_ctime;   // Last inode modification.
    blksize_t  st_blksize; // Block size for I/O operations.
    blkcnt_t   st_blocks;  // File size / block size.
};
```

Va osservato che il file system Minix 1, usato da os32, riporta esclusivamente la data e l'ora di modifica, pertanto le altre due date previste sono sempre uguali a quella di modifica.

Il membro *st_mode*, oltre alla modalità dei permessi che si cambiano con *chmod(2)* [87.7], serve ad annotare altre informazioni. Nel file 'sys/stat.h' sono definite delle macroistruzioni, utili per individuare il tipo di file. Queste macroistruzioni si risolvono in un valore numerico diverso da zero, solo se la condizione che rappresentano è vera:

Macroistruzione	Significato
S_ISBLK (<i>m</i>)	È un file di dispositivo a blocchi?
S_ISCHR (<i>m</i>)	È un file di dispositivo a caratteri?
S_ISFIFO (<i>m</i>)	È un file FIFO?
S_ISREG (<i>m</i>)	È un file puro e semplice?
S_ISDIR (<i>m</i>)	È una directory?
S_ISLNK (<i>m</i>)	È un collegamento simbolico?
S_ISSOCK (<i>m</i>)	È un socket di dominio Unix?

Naturalmente, anche se nel file system possono esistere file di ogni tipo, poi os32 non è in grado di gestire i file FIFO, i collegamenti simbolici e nemmeno i socket di dominio Unix.

Nel file `'sys/stat.h'` sono definite anche delle macro-variabili per individuare e facilitare la selezione dei bit che compongono le informazioni del membro *st_mode*:

Modalità simbolica	Modalità numerica	Descrizione
S_IFMT	0170000 ₈	Maschera che raccoglie tutti i bit che individuano il tipo di file.
S_IFBLK	0060000 ₈	File di dispositivo a blocchi.
S_IFCHR	0020000 ₈	File di dispositivo a caratteri.
S_IFIFO	0010000 ₈	File FIFO.
S_IFREG	0100000 ₈	File puro e semplice.
S_IFDIR	0040000 ₈	Directory.
S_IFLNK	0120000 ₈	Collegamento simbolico.
S_IFSOCK	0140000 ₈	Socket di dominio Unix.

Modalità simbolica	Modalità numerica	Descrizione
S_ISUID	0004000 ₈	SUID.
S_ISGID	0002000 ₈	SGID.
S_ISVTX	0001000 ₈	Sticky.

Modalità simbolica	Modalità numerica	Descrizione
S_IRWXU	0000700 ₈	Lettura, scrittura ed esecuzione per l'utente proprietario.
S_IRUSR	0000400 ₈	Lettura per l'utente proprietario.
S_IWUSR	0000200 ₈	Scrittura per l'utente proprietario.
S_IXUSR	0000100 ₈	Esecuzione per l'utente proprietario.

Modalità simbolica	Modalità numerica	Descrizione
S_IRWXG	0000070 ₈	Lettura, scrittura ed esecuzione per il gruppo.
S_IRGRP	0000040 ₈	Lettura per il gruppo.
S_IWGRP	0000020 ₈	Scrittura per il gruppo.
S_IXGRP	0000010 ₈	Esecuzione per il gruppo.

Modalità simbolica	Modalità numerica	Descrizione
S_IRWXO	0000007 ₈	Lettura, scrittura ed esecuzione per gli altri utenti.
S_IROTH	0000004 ₈	Lettura per gli altri utenti.
S_IWOTH	0000002 ₈	Scrittura per gli altri utenti.
S_IXOTH	0000001 ₈	Esecuzione per gli altri utenti.

os32 non considera i permessi SUID (*Set user id*), SGID (*Set group id*) e Sticky; inoltre, non tiene in considerazione i permessi legati al gruppo, perché non ne tiene traccia.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
ENFILE	Troppi file aperti nel sistema.
ENOENT	File non trovato.
EACCES	Permesso negato.
EBADF	Il descrittore del file richiesto non è valido.

FILE SORGENTI

'lib/sys/stat.h' [95.25]

'lib/sys/stat/stat.c' [95.25.6]

'lib/sys/stat/fstat.c' [95.25.3]

'lib/sys/os32/sys.s' [95.21.7]

'kernel/ibm_i386/isr.s' [94.6.21]

'kernel/proc/sysroutine.c' [94.14.28]

'kernel/lib_s/s_stat.c' [94.8.42]

'kernel/lib_s/s_fstat.c' [94.8.17]

VEDERE ANCHE

chmod(2) [87.7], *chown(2)* [87.8].

87.56 os32: sys(2)

<<

NOME

‘**sys**’ - chiamata di sistema

SINTASSI

```
#include <sys/os32.h>
void sys (int syscallnr, void *message, size_t size);
```

DESCRIZIONE

Attraverso la funzione *sys()* si effettuano tutte le chiamate di sistema, passando al kernel un messaggio, a cui punta *message*, lungo *size* byte. A seconda dei casi, il messaggio può essere modificato dal kernel, come risposta alla chiamata.

Il messaggio è in pratica una variabile strutturata, la cui articolazione cambia a seconda del tipo di chiamata, pertanto si rende necessario specificarne ogni volta la dimensione.

Le strutture usate per comporre i messaggi hanno alcuni membri ricorrenti frequentemente:

Membro	Descrizione
<code>char path[PATH_MAX];</code>	Un percorso di file o directory.
<code>... ret;</code>	Serve a contenere il valore restituito dalla funzione che nel kernel compie effettivamente il lavoro. Il tipo del membro varia caso per caso.
<code>int errno;</code>	Serve a contenere il numero dell'errore prodotto dalla funzione che nel kernel compie effettivamente il lavoro.
<code>int errln;</code>	Serve a contenere il numero della riga di codice in cui si è prodotto un errore.
<code>char errfn[PATH_MAX];</code>	Serve a contenere il nome del file in cui si è prodotto un errore.

Le funzioni che si avvalgono di `sys()`, prima della chiamata devono provvedere a compilare il messaggio con i dati necessari, dopo la chiamata devono acquisire i dati di ritorno, contenuti nel messaggio aggiornato dal kernel, e in particolare devono aggiornare le variabili *errno*, *errln* e *errfl*, utilizzando i membri con lo stesso nome presenti nel messaggio.

FILE SORGENTI

'lib/sys/os32.h' [95.21]

'lib/sys/os32/sys.s' [95.21.7]

'kernel/ibm_i386/isr.s' [94.6.21]

'kernel/proc/sysroutine.c' [94.14.28]

87.57 os32: stime(2)

«
Vedere *time(2)* [87.59].

87.58 os32: tcgetattr(2)

«

NOME

‘**tcgetattr**’, ‘**tcsetattr**’ - lettura o impostazione della configurazione del terminale

SINTASSI

```
#include <termios.h>
int tcgetattr (int fdn, struct termios *termios_p);
int tcsetattr (int fdn, int action,
               struct termios *termios_p);
```

DESCRIZIONE

Le funzioni che fanno capo al file di intestazione ‘`termios.h`’ consentono di gestire la configurazione del terminale, per ciò che riguarda l’input e l’output dello stesso. Va comunque osservato che os32 gestisce il terminale esclusivamente in modalità canonica, sostanzialmente equivalente a quella della vecchia telescrivente.

La configurazione del terminale viene letta o scritta attraverso una variabile strutturata di tipo ‘**struct termios**’, organizzata nel modo seguente:

```

struct termios {
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    cc_t      c_cc[NCCS];
};

```

Il membro *c_cc[]* è un array di caratteri di controllo, a cui viene attribuita una definizione. Il membro *c_iflag* serve a contenere opzioni sull’inserimento, ovvero sul controllo della digitazione. Il membro *c_lflag* serve a contenere delle opzioni definite come «locali», le quali si occupano in pratica di controllare la visualizzazione della digitazione introdotta e di decidere se l’interruzione ricevuta da tastiera debba produrre l’invio di un segnale di interruzione al processo con cui si sta interagendo. Gli altri due membri della struttura non vengono utilizzati da `os32`.

Tabella 84.91. Caratteri di controllo riconosciuti da `os32`, secondo le definizioni del file `termios.h`.

Definizione	Corrispondenza	Descrizione
<i>VEOF</i>	04 ₁₆ <EOT>	Carattere di fine file.
<i>VERASE</i>	08 ₁₆ <BS>	Carattere di cancellazione.
<i>VINTR</i>	03 ₁₆ <ETX>	Carattere di interruzione.
<i>VQUIT</i>	1C ₁₆ <FS>	Carattere di abbandono.

Tabella 84.92. Opzioni del membro *c_iflag* riconosciute da os32.

Opzione	Descrizione
<i>BRKINT</i>	Se questa opzione è attiva e, nel contempo, non è attiva <i>IGNBRK</i> , si intendono recepire i codici di interruzione <i>VINTR</i> . Se l'opzione <i>ISIG</i> del membro <i>c_iflag</i> è attiva, il processo più interno del gruppo a cui appartiene il terminale viene concluso; in ogni caso, viene annullato il contenuto della riga di inserimento in corso.
<i>ICRNL</i>	Se si riceve il carattere <CR> , questo viene convertito in <NL> .
<i>IGNBRK</i>	Se questa opzione è attiva, fa sì che il carattere definito come <i>VINTR</i> sia ignorato.
<i>IGNCR</i>	Se si riceve il carattere <CR> , questo viene ignorato semplicemente.
<i>INLCR</i>	Se si riceve il carattere <NL> , questo viene convertito in <CR> .

Tabella 84.93. Opzioni del membro *c_lflag* riconosciute da *os32*.

Opzione	Descrizione
<i>ECHO</i>	Abilita la visualizzazione sullo schermo del testo inserito da tastiera.
<i>ECHOE</i>	Ammesso che sia attiva l'opzione <i>ECHO</i> , questa abilita il recepimento del carattere definito come <i>VERASE</i> per cancellare l'ultimo carattere inserito, indietreggiando di una posizione.
<i>ECHONL</i>	Indipendentemente dall'opzione <i>ECHO</i> , questa abilita il recepimento del carattere <i><NL></i> per fare avanzare il cursore alla riga successiva, con l'eventuale scorrimento in avanti se si trova già sull'ultima.
<i>ISIG</i>	Ammesso che sia recepito e accettato un codice di interruzione, definito come <i>VINTR</i> , con questa opzione si ottiene l'invio di un segnale di interruzione al processo più interno del gruppo collegato al terminale (il processo più interno dovrebbe corrispondere a quello in primo piano al momento della digitazione).

La funzione *tcgetattr()* legge la configurazione del terminale connesso al descrittore *fdn*, mettendo i dati ottenuti nella struttura a cui punta *termios_p*; al contrario, *tcsetattr()* configura il terminale del descrittore *fdn*, in base ai dati contenuti nella struttura a cui punta *termios_p*.

VALORE RESTITUITO

La funzioni *tcgetattr()* e *tcsetattr()* restituiscono zero in caso di successo, oppure il valore -1 in caso contrario, aggiornando di conseguenza la variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il puntatore <i>termios_p</i> non è valido.
EBADF	Il numero del descrittore di file non è valido.
ENOTTY	Il numero del descrittore di file non corrisponde a un terminale.

DIFETTI

È disponibile soltanto una modalità canonica di funzionamento del terminale.

FILE SORGENTI

‘lib/termios.h’ [95.28]

‘lib/termios/tcgetattr.c’ [95.28.1]

‘lib/termios/tcsetattr.c’ [95.28.2]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

‘kernel/lib_s.h’ [94.8]

‘kernel/lib_s/s_tcgetattr.c’ [94.8.44]

‘kernel/lib_s/s_tcsetattr.c’ [94.8.45]

VEDERE ANCHE

tty(1) [86.27], *isatty(3)* [88.69].

87.59 os32: time(2)



NOME

‘**time**’, ‘**stime**’ - lettura o impostazione della data e dell’ora del sistema

SINTASSI

```
#include <time.h>
time_t time (time_t *timer);
int      stime (time_t *timer);
```

DESCRIZIONE

La funzione *time()* legge la data e l’ora attuale del sistema, espressa in secondi; se il puntatore *timer* è valido (non è ‘**NULL**’), il risultato dell’interrogazione viene salvato anche in ciò a cui questo punta.

La funzione *stime()* consente di modificare la data e l’ora attuale del sistema, fornendo il puntatore alla variabile contenente la quantità di secondi trascorsi a partire dall’ora zero del 1 gennaio 1970.

VALORE RESTITUITO

La funzione *time()* restituisce la data e l’ora attuale del sistema, espressa in secondi trascorsi a partire dall’ora zero del 1 gennaio 1970.

La funzione *stime()* restituisce zero in caso di successo e, teoricamente, -1 in caso di errore, ma attualmente nessun tipo di errore è previsto.

DIFETTI

La funzione *stime()* dovrebbe essere riservata a un utente privilegiato, mentre attualmente qualunque utente può servirsene per cambiare la data di sistema.

FILE SORGENTI

‘lib/time.h’ [95.29]
‘lib/time/time.c’ [95.29.6]
‘lib/time/stime.c’ [95.29.5]
‘lib/sys/os32/sys.s’ [95.21.7]
‘kernel/ibm_i386/isr.s’ [94.6.21]
‘kernel/proc/sysroutine.c’ [94.14.28]
‘kernel/lib_k.h’ [94.7]
‘kernel/lib_s/s_time.c’ [94.8.46]
‘kernel/lib_s/s_stime.c’ [94.8.43]

VEDERE ANCHE

date(1) [86.10], *ctime(3)* [88.15].

87.60 os32: umask(2)

«

NOME

‘**umask**’ - maschera dei permessi

SINTASSI

```
#include <sys/stat.h>
mode_t umask (mode_t mask);
```

DESCRIZIONE

La funzione *umask()* modifica la maschera dei permessi associata al processo in corso. Nel contenuto del parametro *mask* vengono presi in considerazione soltanto i nove bit meno significativi, i quali rappresentano i permessi di accesso di utente proprietario, gruppo e altri utenti.

La maschera dei permessi viene usata dalle funzioni che creano dei file, per limitare i permessi in modo automatico: ciò che appare attivo nella maschera è quello che non viene consentito nella creazione del file.

VALORE RESTITUITO

La funzione restituisce il valore che aveva la maschera dei permessi prima della chiamata.

FILE SORGENTI

‘lib/sys/stat.h’ [95.25]

‘lib/sys/stat/umask.c’ [95.25.7]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

VEDERE ANCHE

mkdir(2) [87.34], *chmod(2)* [87.7], *open(2)* [87.37], *stat(2)* [87.55].

87.61 os32: umount(2)

«
Vedere *mount(2)* [87.36].

87.62 os32: unlink(2)

«

NOME

‘**unlink**’ - cancellazione di un nome

SINTASSI

```
#include <unistd.h>
int unlink (const char *path);
```

DESCRIZIONE

La funzione *unlink()* cancella un nome da una directory, ma se si tratta dell'ultimo collegamento che ha quel file, allora libera anche l'inode corrispondente.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
-1	Errore: la variabile <i>errno</i> viene impostata di conseguenza.

ERRORI

Valore di <i>errno</i>	Significato
ENOTEMPTY	È stata tentata la cancellazione di una directory, ma questa non è vuota.
ENOTDIR	Una delle directory del percorso, non è una directory.
ENOENT	Il nome richiesto non esiste.
EROFS	Il file system è in sola lettura.
EPERM	Mancano i permessi necessari.
EUNKNOWN	Si è verificato un errore imprevisto e sconosciuto.

FILE SORGENTI

‘lib/unistd.h’ [95.30]

‘lib/unistd/unlink.c’ [95.30.42]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

‘kernel/lib_s/s_unlink.c’ [94.8.48]

VEDERE ANCHE

rm(1) [86.22], *link*(2) [87.30], *rmdir*(2) [87.41].

87.63 os32: wait(2)

NOME

‘wait’ - attesa della morte di un processo figlio

SINTASSI

```
#include <sys/wait.h>
pid_t wait (int *status);
```

DESCRIZIONE

La funzione *wait()* mette il processo in pausa, in attesa della morte di un processo figlio; quando ciò dovesse accadere, il valore di **status* verrebbe aggiornato con quanto restituito dal processo defunto e il processo sospeso riprenderebbe l'esecuzione.

VALORE RESTITUITO

La funzione restituisce il numero del processo defunto, oppure -1 se non ci sono processi figli.

ERRORI

Valore di <i>errno</i>	Significato
ECHILD	Non ci sono processi figli da attendere.

FILE SORGENTI

'lib/sys/wait.h' [[95.27](#)]

'lib/sys/wait/wait.c' [[95.27.1](#)]

'lib/sys/os32/sys.s' [[95.21.7](#)]

'kernel/ibm_i386/isr.s' [[94.6.21](#)]

'kernel/proc/sysroutine.c' [[94.14.28](#)]

'kernel/lib_s/s_wait.c' [[94.8.49](#)]

VEDERE ANCHE

_exit(2) [[87.2](#)], *fork(2)* [[87.19](#)], *kill(2)* [[87.29](#)], *signal(2)* [[87.52](#)].

87.64 os32: write(2)



NOME

‘**write**’ - scrittura di un descrittore di file

SINTASSI

```
#include <unistd.h>
ssize_t write (int fdn, const void *buffer, size_t count);
```

DESCRIZIONE

La funzione *write()* consente di scrivere fino a un massimo di *count* byte, tratti dall’area di memoria che inizia all’indirizzo *buffer*, presso il file rappresentato dal descrittore *fdn*. La scrittura avviene a partire dalla posizione in cui si trova l’indice interno.

VALORE RESTITUITO

La funzione restituisce la quantità di byte scritti effettivamente e in tal caso è possibile anche ottenere una quantità pari a zero. Se si verifica invece un errore, la funzione restituisce -1 e aggiorna la variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il numero del descrittore di file non è valido.
EINVAL	Il file non è aperto in scrittura.
EISDIR	Il file è una directory.
E_FILE_TYPE_UNSUPPORTED	Il file è di tipo non gestibile con os32.
EIO	Errore di input-output.

FILE SORGENTI

‘lib/unistd.h’ [95.30]

‘lib/unistd/write.c’ [95.30.43]

‘lib/sys/os32/sys.s’ [95.21.7]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

‘kernel/lib_s/s_write.c’ [94.8.50]

VEDERE ANCHE

close(2) [87.10], *lseek(2)* [87.33], *open(2)* [87.37], *read(2)* [87.39], *fwrite(3)* [88.49].

87.65 os32: z(2)

«

NOME

‘z_...’ - funzioni provvisorie

SINTASSI

```
#include <sys/os32.h>
void z_perror    (const char *string);
int  z_printf   (char *format, ...);
int  z_vprintf  (char *format, va_list arg);
```

DESCRIZIONE

Le funzioni del gruppo ‘**z_...()**’ eseguono compiti equivalenti a quelli delle funzioni di libreria con lo stesso nome, ma prive del prefisso ‘**z_**’. Queste funzioni ‘**z_...()**’ si avvalgono, per il loro lavoro, di chiamate di sistema particolari; la loro realizzazione si è resa necessaria durante lo sviluppo di os32, prima che potesse essere disponibile un sistema di gestione centralizzato dei dispositivi.

Queste funzioni non sono più utili, ma rimangono per documentare le fasi realizzative iniziali di os32 e, d’altro canto, possono servire se si rende necessario aggirare la gestione dei dispositivi per visualizzare dei messaggi sullo schermo.

FILE SORGENTI

‘lib/sys/os32.h’ [[95.21](#)]

‘lib/sys/os32/z_perror.c’ [[95.21.9](#)]

‘lib/sys/os32/z_printf.c’ [[95.21.10](#)]

‘lib/sys/os32/z_vprintf.c’ [[95.21.11](#)]

VEDERE ANCHE

perror(3) [[88.90](#)], *printf(3)* [[88.91](#)], *putchar(3)* [[88.38](#)], *puts(3)* [[88.39](#)], *vprintf(3)* [[88.137](#)], *vsprintf(3)* [[88.137](#)].

87.66 os32: z_perror(2)

«
Vedere `z(2)` [[87.65](#)].

87.67 os32: z_printf(2)

«
Vedere `z(2)` [[87.65](#)].

87.68 os32: z_vprintf(2)

«
Vedere `z(2)` [[87.65](#)].

Sezione 3: funzioni di libreria

88.1 os32: _gcc(3)

NOME

‘_gcc’ - libreria per il compilatore

DESCRIZIONE

Le funzioni descritte dal file di intestazione ‘lib/gcc.h’ e contenute nella directory ‘lib/_gcc/’, servono al compilatore GNU C per compiere il proprio lavoro correttamente con valori da 64 bit.

FILE SORGENTI

‘lib/_gcc.h’ [95.2].

88.2 os32: abort(3)

NOME

‘abort’ - conclusione anormale del processo

SINTASSI

```
#include <stdlib.h>
void abort (void);
```

DESCRIZIONE

La funzione *abort()* verifica lo stato di configurazione del segnale ‘SIGABRT’ e, se risulta bloccato, lo sblocca, quindi invia questo segnale per il processo in corso. Ciò provoca la conclusione del

processo, secondo la modalità prevista per tale segnale, a meno che il segnale sia stato ridiretto a una funzione, nel qual caso, dopo l'invio del segnale, potrebbe esserci anche una ripresa del controllo da parte della funzione *abort()*. Tuttavia, se così fosse, il segnale '**SIGABRT**' verrebbe poi riconfigurato alla sua impostazione normale e verrebbe inviato nuovamente lo stesso segnale per provocare la conclusione del processo. Pertanto, la funzione *abort()* non restituisce il controllo.

Va comunque osservato che os32 non è in grado di associare una funzione a un segnale, pertanto, i segnali possono solo avere una gestione predefinita, o al massimo risultare bloccati.

FILE SORGENTI

'lib/stdlib.h' [95.19]

'lib/stdlib/abort.c' [95.19.2]

VEDERE ANCHE

signal(2) [87.52].

88.3 os32: abs(3)

«

NOME

'abs', 'labs', 'llabs', 'imaxabs' - valore assoluto di un numero intero

SINTASSI

```
#include <stdlib.h>
int abs (int j);
long int labs (long int j);
long long int llabs (long long int j);
```

```
#include <inttypes.h>
intmax_t imaxabs (intmax_t j);
```

DESCRIZIONE

Le funzioni ‘...**abs** ()’ restituiscono il valore assoluto del loro argomento, distinguendosi per tipo di intero.

VALORE RESTITUITO

Il valore assoluto del numero intero fornito come argomento.

FILE SORGENTI

‘lib/stdlib.h’ [95.19]

‘lib/stdlib/abs.c’ [95.19.3]

‘lib/stdlib/labs.c’ [95.19.11]

‘lib/stdlib/llabs.c’ [95.19.13]

‘lib/inttypes.h’ [95.8]

‘lib/inttypes/imaxabs.c’ [95.8.1]

VEDERE ANCHE

div(3) [88.17], *ldiv(3)* [88.17], *lldiv(3)* [88.17], *imaxdiv(3)* [88.17], *rand(3)* [88.97].

88.4 os32: access(3)

NOME

‘**access**’ - verifica dei permessi di accesso dell’utente

SINTASSI

```
#include <unistd.h>
int access (const char *path, int mode);
```

DESCRIZIONE

La funzione *access()* verifica lo stato di accessibilità del file indicato nella stringa *path*, secondo i permessi stabiliti con il parametro *mode*.

L'argomento corrispondente al parametro *mode* può assumere un valore corrispondente alla macro-variabile *F_OK*, per verificare semplicemente l'esistenza del file specificato; altrimenti, può avere un valore composto dalla combinazione (con l'operatore OR binario) di *R_OK*, *W_OK* e *X_OK*, per verificare, rispettivamente, l'accessibilità in lettura, in scrittura e in esecuzione. Queste macro-variabili sono dichiarate nel file 'unistd.h'.

VALORE RESTITUITO

Valore	Significato
0	I permessi di accesso richiesti sono tutti disponibili.
-1	I permessi non sono tutti disponibili, oppure si è verificato un errore di altro genere, da chiarire analizzando la variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
ENFILE	Troppi file aperti nel sistema.
ENOENT	File non trovato.
EACCES	Permesso negato.

DIFETTI

Questa realizzazione della funzione *access()* determina l'accessibilità a un file attraverso le informazioni che può trarre autonomamente; pertanto, si tratta di una valutazione presunta e non reale.

FILE SORGENTI

'lib/unistd.h' [95.30]

'lib/unistd/access.c' [95.30.2]

VEDERE ANCHE

stat(2) [87.55].

88.5 os32: *asctime(3)*

Vedere *ctime(3)* [88.15].

88.6 os32: *assert(3)*

NOME

'**assert**' - verifica diagnostica del risultato di un'espressione

SINTASSI

```
#include <assert.h>
void assert (tipo_scalare espressione);
```

DESCRIZIONE

Il file ‘`assert.h`’ della libreria standard definisce la macroistruzione ***assert()***, da usare per generare informazioni diagnostiche, sulla base dell’esito della valutazione di un’espressione.

La macroistruzione ***assert()*** viene definita in due modi alternativi, in base alla presenza o meno della macro-variabile ***NDEBUG***. Per la precisione, in presenza della macro-variabile ***NDEBUG*** la macroistruzione ***assert()*** risulta inerte.

La macroistruzione ***assert()*** va usata con la sintassi indicata, dove il parametro indica un’espressione di tipo non specificato, purché di tipo scalare. Se l’espressione si traduce in un valore *Falso*, ovvero pari a zero, la macroistruzione emette, attraverso lo standard error, un messaggio contenente l’espressione stessa e altre indicazioni. Precisamente, oltre all’espressione appaiono: il nome della funzione in cui ci si trova, il nome del file (sorgente) e il numero della riga.

FILE SORGENTI

‘`lib/assert.h`’ [[95.1.3](#)]

88.7 os32: atexit(3)

«

NOME

‘**atexit**’, ‘**exit**’ - gestione della chiusura dei processi

SINTASSI

```
#include <stdlib.h>
typedef void (*atexit_t) (void);
int  atexit (atexit_t function);
void exit   (int status);
```

DESCRIZIONE

La funzione *exit()* conclude il processo in corso, avvalendosi della chiamata di sistema *_exit(2)* [87.2], ma prima di farlo, scandisce un array contenente un elenco di funzioni, da eseguire prima di tale chiamata finale. Questo array viene popolato eventualmente con l'aiuto della funzione *atexit()*, sapendo che l'ultima funzione di chiusura aggiunta, è la prima a dover essere eseguita alla conclusione.

La funzione *atexit()* riceve come argomento il puntatore a una funzione che non prevede argomenti e non restituisce alcunché. Per facilitare la dichiarazione del prototipo e, di conseguenza, dell'array usato per accumulare tali puntatori, il file di intestazione 'stdlib.h' di os32 dichiara un tipo speciale, non standard, denominato 'atexit_t', definito come:

```
typedef void (*atexit_t) (void);
```

Si possono annotare un massimo di *ATEXIT_MAX* funzioni da eseguire prima della conclusione di un processo. Tale macro-variabile è definita nel file 'limits.h'.

VALORE RESTITUITO

Solo la funzione *atexit()* restituisce un valore, perché *exit()* non può nemmeno restituire il controllo.

Valore	Significato
0	Operazione conclusa con successo.
-1	Errore, dovuto all'esaurimento dello spazio nell'array usato per accumulare le funzioni di chiusura.

FILE SORGENTI

'lib/limits.h' [95.1.6]

'lib/stdlib.h' [95.19]

'lib/stdlib/atexit.c' [95.19.4]

'lib/stdlib/exit.c' [95.19.9]

VEDERE ANCHE

`_exit(2)` [87.2], `_Exit(2)` [87.2].

88.8 os32: atoi(3)

«

NOME

'**atoi**', '**atol**' - conversione da stringa a numero intero

SINTASSI

```
#include <stdlib.h>
int atoi (const char *string);
long int atol (const char *string);
```

DESCRIZIONE

Le funzioni '**ato...()**' convertono una stringa, fornita come argomento, in un numero intero. La conversione avviene escludendo gli spazi iniziali, considerando eventualmente un segno («+» o «-») e poi soltanto i caratteri che rappresentano cifre numeriche.

La scansione della stringa e l'interpretazione del valore numerico contenuto terminano quando si incontra un carattere diverso dalle cifre numeriche.

VALORE RESTITUITO

Il valore numerico ottenuto dall'interpretazione della stringa.

FILE SORGENTI

'lib/stdlib.h' [95.19]

'lib/stdlib/atoi.c' [95.19.5]

'lib/stdlib/atol.c' [95.19.6]

VEDERE ANCHE

strtol(3) [88.130], *strtoul(3)* [88.130].

88.9 os32: atol(3)

Vedere *atoi(3)* [88.8].

88.10 os32: basename(3)

NOME

'**basename**', '**dirname**' - elaborazione dei componenti di un percorso

SINTASSI

```
#include <libgen.h>
char *basename (char *path);
char *dirname (char *path);
```

DESCRIZIONE

Le funzioni *basename()* e *dirname()*, restituiscono un percorso, estratto da quello fornito come argomento (*path*). Per la precisione, *basename()* restituisce l'ultimo componente del percorso, mentre *dirname()* restituisce ciò che precede l'ultimo componente. Valgono gli esempi seguenti:

Contenuto originale di <i>path</i>	Risultato prodotto da ' <i>dirname (path)</i> '	Risultato prodotto da ' <i>basename (path)</i> '
"/usr/bin/	"/usr"	"bin"
"/usr/bin	"/usr"	"bin"
"/usr	"/"	"usr"
"usr	"."	"usr"
"/"	"/"	"/"
"."	"."	"."
".."	".."	".."

È importante considerare che le due funzioni alterano il contenuto di *path*, in modo da isolare i componenti che servono.

VALORE RESTITUITO

Le due funzioni restituiscono il puntatore alla stringa contenente il risultato dell'elaborazione, trattandosi di una porzione della stringa già usata come argomento della chiamata e modificata per l'occasione. Non è previsto il manifestarsi di alcun errore.

FILE SORGENTI

‘lib/libgen.h’ [95.9]

‘lib/libgen/basename.c’ [95.9.1]

‘lib/libgen/dirname.c’ [95.9.2]

88.11 os32: byteorder(3)



NOME

‘htonl’, ‘htons’, ‘ntohl’, ‘ntohs’ - conversione dell’ordine dei byte da *host* a *network* e viceversa

SINTASSI

```
#include <arpa/inet.h>
uint32_t htonl (uint32_t host32);
uint16_t htons (uint16_t host16);
uint32_t ntohl (uint32_t net32);
uint16_t ntohs (uint16_t net16);
```

DESCRIZIONE

La funzione *htonl()* converte un valore a 32 bit, dall’ordinamento di byte usato dal sistema, nell’ordinamento adatto alla trasmissione in rete: *Host to network long*.

La funzione *htons()* converte un valore a 16 bit, dall’ordinamento di byte usato dal sistema, nell’ordinamento adatto alla trasmissione in rete: *Host to network short*.

La funzione *ntohl()* converte un valore a 32 bit, dall’ordinamento di byte adatto alla trasmissione in rete, nell’ordinamento usato nel sistema: *Network to host long*.

La funzione *ntohs()* converte un valore a 16 bit, dall'ordinamento di byte adatto alla trasmissione in rete, nell'ordinamento usato nel sistema: *Network to host short*.

In un sistema os32, l'ordine dei byte è tale da avere prima il byte meno significativo, mentre l'ordine usato nella trasmissione in rete richiede di avere prima il byte più significativo.

FILE SORGENTI

'lib/arpa/inet.h' [95.3]

'lib/arpa/inet/htonl.c' [95.3.1]

'lib/arpa/inet/htons.c' [95.3.2]

'lib/arpa/inet/ntohl.c' [95.3.5]

'lib/arpa/inet/ntohs.c' [95.3.6]

VEDERE ANCHE

inet_ntop(3) [88.66], *inet_pton(3)* [88.67].

88.12 os32: clearerr(3)

«

NOME

'**clearerr**' - azzeramento degli indicatori di errore e di fine file di un certo flusso di file

SINTASSI

```
#include <stdio.h>
void clearerr (FILE *fp);
```

DESCRIZIONE

La funzione *clearerr()* azzerava gli indicatori di errore e di fine file, del flusso di file indicato come argomento.

FILE SORGENTI

‘lib/stdio.h’ [95.18]

‘lib/stdio/clearerr.c’ [95.18.2]

VEDERE ANCHE

feof(3) [88.29], *ferror(3)* [88.30], *fileno(3)* [88.35], *stdio(3)* [88.112].

88.13 os32: closedir(3)



NOME

‘**closedir**’ - chiusura di una directory

SINTASSI

```
#include <sys/types.h>
#include <dirent.h>
int closedir (DIR *dp);
```

DESCRIZIONE

La funzione *closedir()* chiude la directory rappresentata da *dp*.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
-1	Errore: la variabile <i>errno</i> viene impostata di conseguenza.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	La directory rappresentata da <i>dp</i> , non è valida.

FILE SORGENTI

‘lib/sys/types.h’ [95.26]

‘lib/dirent.h’ [95.4]

‘lib/dirent/DIR.c’ [95.4.1]

‘lib/dirent/closedir.c’ [95.4.2]

VEDERE ANCHE

close(2) [87.10], *opendir(3)* [88.89], *readdir(3)* [88.98],
rewinddir(3) [88.101].

88.14 os32: creat(3)

«

NOME

‘**creat**’ - creazione di un file puro e semplice

SINTASSI

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *path, mode_t mode);
```

DESCRIZIONE

La funzione *creat()* equivale esattamente all’uso della funzione *open()*, con le opzioni ‘O_WRONLY|O_CREAT|O_TRUNC’:

```
open (path, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

Per ogni altra informazione, si veda la pagina di manuale *open(2)* [87.37].

FILE SORGENTI

‘lib/sys/types.h’ [95.26]

‘lib/sys/stat.h’ [95.25]

‘lib/fcntl.h’ [95.6]

‘lib/fcntl/creat.c’ [95.6.1]

VEDERE ANCHE

chmod(2) [87.7], *chown(2)* [87.8], *close(2)* [87.10], *dup(2)* [87.12], *fcntl(2)* [87.18], *link(2)* [87.33], *mknod(2)* [87.35], *mount(2)* [87.36], *open(2)* [87.37], *read(2)* [87.39], *stat(2)* [87.55], *umask(2)* [87.60], *unlink(2)* [87.62], *write(2)* [87.64], *fopen(3)* [88.36].

88.15 os32: ctime(3)



NOME

‘**asctime**’, ‘**ctime**’, ‘**gmtime**’, ‘**localtime**’, ‘**mktime**’ -
conversione di informazioni data-orario

SINTASSI

```
#include <time.h>
char      *asctime      (const struct tm *timeptr);
char      *ctime       (const time_t *timer);
struct tm *gmtime      (const time_t *timer);
struct tm *localtime   (const time_t *timer);
time_t    mktime       (const struct tm *timeptr);
```

DESCRIZIONE

Queste funzioni hanno in comune il compito di convertire delle informazioni data-orario, da un formato a un altro, eventualmente anche testuale. Una data e un orario possono essere rappresentati con il tipo `'time_t'`, nel qual caso si tratta del numero di secondi trascorsi dall'ora zero del 1 gennaio 1970; in alternativa potrebbe essere rappresentata in una variabile strutturata di tipo `'struct tm'`, dichiarato nel file `'time.h'`:

```
struct tm {
    int tm_sec;      // secondi
    int tm_min;     // minuti
    int tm_hour;    // ore
    int tm_mday;    // giorno del mese
    int tm_mon;     // mese, da 1 a 12
    int tm_year;    // anno
    int tm_wday;    // giorno della settimana,
                    // da 0 (domenica) a 6
    int tm_yday;    // giorno dell'anno
    int tm_isdst;   // informazioni sull'ora estiva
};
```

In alcuni casi, la conversione dovrebbe tenere conto della configurazione locale, ovvero del fuso orario ed eventualmente del cambiamento di orario nel periodo estivo. `os32` non considera alcunché e gestisce il tempo in un modo assoluto, senza nozione della convenzione locale.

La funzione `asctime()` converte quanto contenuto in una variabile strutturata di tipo `'struct tm'` in una stringa che descrive la data e l'ora in inglese. La stringa in questione è allocata staticamente e viene sovrascritta se la funzione viene usata più volte.

La funzione `ctime()` è in realtà soltanto una macroistruzione, la

quale, complessivamente, converte il tempo indicato come quantità di secondi, restituendo il puntatore a una stringa che descrive la data attuale, tenendo conto della configurazione locale. La stringa in questione utilizza la stessa memoria statica usata per *asctime()*; inoltre, dato che `os32` non distingue tra ora locale e tempo universale, la funzione non esegue alcuna conversione temporale.

La funzione *gmtime()* converte il tempo espresso in secondi in una forma suddivisa, secondo il tipo `'struct tm'`. La funzione restituisce quindi il puntatore a una variabile strutturata di tipo `'struct tm'`, la quale però è dichiarata in modo statico, internamente alla funzione, e viene sovrascritta nelle chiamate successive della stessa.

La funzione *localtime()* converte una data espressa in secondi, in una data suddivisa in campi (`'struct tm'`), tenendo conto (ma non succede con `os32`) della configurazione locale.

La funzione *mktime()* converte una data contenute in una variabile strutturata di tipo `'struct tm'` nella quantità di secondi corrispondente.

VALORE RESTITUITO

Le funzioni che restituiscono un puntatore, se incontrano un errore, restituiscono il puntatore nullo, `'NULL'`. Nel caso particolare di *mktime()*, se il valore restituito è pari a `-1`, si tratta di un errore.

FILE SORGENTI

`'lib/time.h'` [95.29]

`'lib/time/asctime.c'` [95.29.1]

`'lib/time/gmtime.c'` [95.29.3]

'lib/time/mktime.c' [95.29.4]

VEDERE ANCHE

date(1) [86.10], *clock(2)* [87.9], *time(2)* [87.59].

88.16 os32: *dirname(3)*

<<

Vedere *basename(3)* [88.10].

88.17 os32: *div(3)*

<<

NOME

'**div**', '**ldiv**', '**lldiv**', '**imaxdiv**' - calcolo del quoziente e del resto di una divisione intera

SINTASSI

```
#include <stdlib.h>
div_t div (int numer, int denom);
ldiv_t ldiv (long int numer, long int denom);
lldiv_t lldiv (long long int numer, long long int denom);
```

```
#include <inttypes.h>
imaxdiv_t imaxdiv (intmax_t numer, intmax_t denom);
```

DESCRIZIONE

Le funzioni '**...div()**' calcolano la divisione tra numeratore e denominatore, forniti come argomenti della chiamata, restituendo un risultato, composto di divisione intera e resto, in una variabile strutturata.

I tipi `'div_t'`, `'ldiv_t'` e `'lldiv_t'`, sono dichiarati nel file `'stdlib.h'` nel modo seguente:

```
typedef struct {
    int    quot;
    int    rem;
} div_t;
```

```
typedef struct {
    long long int quot;
    long long int rem;
} lldiv_t;
```

```
typedef struct {
    long long int quot;
    long long int rem;
} lldiv_t;
```

Il tipo `'imaxdiv_t'` è dichiarati nel file `'inttypes.h'` in maniera analoga:

```
typedef struct {
    intmax_t quot;
    intmax_t rem;
} imaxdiv_t;
```

I membri *quot* contengono il quoziente, ovvero il risultato intero; i membri *rem* contengono il resto della divisione.

VALORE RESTITUITO

Il risultato della divisione, strutturato in quoziente e resto.

FILE SORGENTI

`'lib/stdlib.h'` [[95.19](#)]

`'lib/stdlib/div.c'` [[95.19.7](#)]

`'lib/stdlib/ldiv.c'` [[95.19.12](#)]

'lib/stdlib/lldiv.c' [95.19.14]

'lib/inttypes.h' [95.8]

'lib/inttypes/imaxdiv.c' [95.8.2]

VEDERE ANCHE

abs(3) [88.3].

88.18 os32: *endgrent(3)*

«

Vedere *getgrent(3)* [88.53].

88.19 os32: *endpwent(3)*

«

Vedere *getpwent(3)* [88.57].

88.20 os32: *errno(3)*

«

NOME

'**errno**' - numero dell'ultimo errore riportato

SINTASSI

```
#include <errno.h>
```

DESCRIZIONE

Attraverso l'inclusione del file 'errno.h', si ottiene la dichiarazione della variabile *errno*. In pratica, per os32 viene dichiarata così:

```
extern int errno;
```

Per annotare un errore si assegna un valore numerico a questa variabile. Il valore numerico in questione rappresenta sinteticamente la descrizione dell'errore; pertanto, si utilizzano per questo delle macro-variabili, dichiarate tutte nel file `errno.h`. Nell'esempio seguente si annota l'errore **ENOMEM**, corrispondente alla descrizione «Not enough space», ovvero «spazio insufficiente»:

```
errno = ENOMEM;
```

Dal momento che in questo modo viene comunque a mancare un riferimento al sorgente e alla posizione in cui l'errore si è manifestato, la libreria di `os32` aggiunge la macroistruzione **errset()**, la quale però non fa parte dello standard. Si usa così:

```
errset (ENOMEM) ;
```

La macroistruzione **errset()** aggiorna la variabile **errln** e l'array **errfn[]**, rispettivamente con il numero della riga di codice in cui si è manifestato il problema e il nome della funzione che lo contiene. Con queste informazioni, la funzione **perror(3)** [88.90] può visualizzare più dati.

Pertanto, nel codice di `os32`, si usa sempre la macroistruzione **errset()**, invece di assegnare semplicemente un valore alla variabile **errno**.

ERRORI

Gli errori previsti dalla libreria di `os32` sono riassunti dalla tabella successiva. La prima parte contiene gli errori definiti dallo standard POSIX, ma solo alcuni di questi vengono usati effettivamente nella libreria, data la limitatezza di `os32`.

Valore di <i>errno</i>	Definizione
E2BIG	Argument list too long.
EACCES	Permission denied.
EADDRINUSE	Address in use.
EADDRNOTAVAIL	Address not available.
EAFNOSUPPORT	Address family not supported.

Valore di <i>errno</i>	Definizione
EAGAIN	Resource unavailable, try again.
EALREADY	Connection already in progress.
EBADF	Bad file descriptor.
EBADMSG	Bad message.
EBUSY	Device or resource busy.

Valore di <i>errno</i>	Definizione
ECANCELED	Operation canceled.
ECHILD	No child processes.
ECONNABORTED	Connection aborted.
ECONNREFUSED	Connection refused.
ECONNRESET	Connection reset.

Valore di <i>errno</i>	Definizione
EDEADLK	Resource deadlock would occur.
EDESTADDRREQ	Destination address required.
EDOM	Mathematics argument out of domain of function.
EDQUOT	Reserved.
EEXIST	File exists.

Valore di <i>errno</i>	Definizione
EFAULT	Bad address.
EFBIG	File too large.
EHOSTUNREACH	Host is unreachable.
EIDRM	Identifier removed.
EILSEQ	Illegal byte sequence.

Valore di <i>errno</i>	Definizione
EINPROGRESS	Operation in progress.
EINTR	Interrupted function.
EINVAL	Invalid argument.
EIO	I/O error.
EISCONN	Socket is connected.

Valore di <i>errno</i>	Definizione
EISDIR	Is a directory.
ELOOP	Too many levels of symbolic links.
EMFILE	Too many open files.
EMLINK	Too many links.
EMSGSIZE	Message too large.

Valore di <i>errno</i>	Definizione
EMULTIHOP	Reserved.
ENAMETOOLONG	Filename too long.
ENETDOWN	Network is down.
ENETRESET	Connection aborted by network.
ENETUNREACH	Network unreachable.

Valore di <i>errno</i>	Definizione
ENFILE	Too many files open in system.
ENOBUFFS	No buffer space available.
ENODATA	No message is available on the stream head read queue.
ENODEV	No such device.
ENOENT	No such file or directory.

Valore di <i>errno</i>	Definizione
ENOEXEC	Executable file format error.
ENOLCK	No locks available.
ENOLINK	Reserved.
ENOMEM	Not enough space.
ENOMSG	No message of the desired type.

Valore di <i>errno</i>	Definizione
ENOPROTOPT	Protocol not available.
ENOSPC	No space left on device.
ENOSR	No stream resources.
ENOSTR	Not a stream.
ENOSYS	Function not supported.

Valore di <i>errno</i>	Definizione
ENOTCONN	The socket is not connected.
ENOTDIR	Not a directory.
ENOTEMPTY	Directory not empty.
ENOTSOCK	Not a socket.
ENOTSUP	Not supported.

Valore di <i>errno</i>	Definizione
ENOTTY	Inappropriate I/O control operation.
ENXIO	No such device or address.
EOPNOTSUPP	Operation not supported on socket.
EOVERFLOW	Value too large to be stored in data type.
EPERM	Operation not permitted.

Valore di <i>errno</i>	Definizione
EPIPE	Broken pipe.
EPROTO	Protocol error.
EPROTONOSUPPORT	Protocol not supported.
EPROTOTYPE	Protocol wrong type for socket.
ERANGE	Result too large.

Valore di <i>errno</i>	Definizione
EROFS	Read-only file system.
ESPIPE	Invalid seek.
ESRCH	No such process.
ESTALE	Reserved.
ETIME	Stream ioctl() timeout.

Valore di <i>errno</i>	Definizione
ETIMEDOUT	Connection timed out.
ETXTBSY	Text file busy.
EWOULDBLOCK	Operation would block (may be the same as EAGAIN).
EXDEV	Cross-device link.

La tabella successiva raccoglie le definizioni degli errori aggiuntivi, specifici di os32.

Valore di <i>errno</i>	Definizione
EUNKNOWN	Unknown error.
E_NO_MEDIUM	No medium found.
E_MEDIUM	Medium reported error.
E_FILE_TYPE	File type not compatible.
E_ROOT_INODE_NOT_CACHED	The root directory inode is not cached.

Valore di <i>errno</i>	Definizione
E_CANNOT_READ_SUPERBLOCK	Cannot read super block.
E_MAP_INODE_TOO_BIG	Map inode too big.
E_MAP_ZONE_TOO_BIG	Map zone too big.
E_DATA_ZONE_TOO_BIG	Data zone too big.
E_CANNOT_FIND_ROOT_DEVICE	Cannot find root device.

Valore di <i>errno</i>	Definizione
E_CANNOT_FIND_ROOT_INODE	Cannot find root inode.
E_FILE_TYPE_UNSUPPORTED	File type unsupported.
E_ENV_TOO_BIG	Environment too big.
E_LIMIT	Exceeded implementation limits.
E_NOT_MOUNTED	Not mounted.

Valore di <i>errno</i>	Definizione
E_NOT_IMPLEMENTED	Not implemented.
E_HARDWARE_FAULT	Hardware fault.
E_DRIVER_FAULT	Driver fault.
E_PIPE_FULL	Pipe full.
E_PIPE_EMPTY	Pipe empty.

FILE SORGENTI

‘lib/errno.h’ [95.5]

‘lib/errno/errno.c’ [95.5.1]

VEDERE ANCHE

perror(3) [88.90], *strerror(3)* [88.120].

88.21 os32: exec(3)



NOME

‘**execl**’, ‘**execle**’, ‘**execlp**’, ‘**execv**’, ‘**execvp**’ - esecuzione di un file

SINTASSI

```
#include <unistd.h>
extern char **environ;
int execl (const char *path, const char *arg, ...);
int execle (const char *path, const char *arg, ...);
int execlp (const char *path, const char *arg, ...);
int execv (const char *path, char *const argv[]);
int execvp (const char *path, char *const argv[]);
```

DESCRIZIONE

Le funzioni ‘**exec... ()**’ rimpiazzano il processo chiamante con un altro processo, ottenuto dal caricamento di un file eseguibile in memoria. Tutte queste funzioni si avvalgono, direttamente o indirettamente di *execve(2)* [87.14].

Il primo parametro delle funzioni descritte qui rappresenta il percorso, costituito dalla stringa *path*, di un file da eseguire.

Le funzioni ‘**execl... ()**’, dopo il percorso del file eseguibile, richiedono l’indicazione di una quantità variabile di argomenti, a cominciare da *arg*, costituiti da stringhe, tenendo conto che dopo l’ultimo di questi argomenti, occorre fornire il puntatore nullo, ‘**NULL**’, per chiarire che questi sono terminati. L’argomento corrispondente al parametro *arg* deve essere una stringa contenente

il nome del file da avviare, mentre quelli successivi sono a loro volta gli argomenti da passare al programma stesso.

Rispetto al gruppo di funzioni ‘**exec1... ()**’, la funzione *execle()*, dopo il puntatore nullo che conclude la sequenza di argomenti per il programma da avviare, si attende una sequenza di altre stringhe, anche questa conclusa da un ultimo e definitivo puntatore nullo. Questa ulteriore sequenza di stringhe va a costituire l’ambiente del processo da avviare, pertanto il contenuto di tali stringhe deve essere del tipo ‘*nome=valore*’.

Le funzioni ‘**execv... ()**’ hanno come ultimo parametro il puntatore a un array di stringhe, dove *argv[0]* deve essere il nome del file da avviare, mentre da *argv[1]* in poi, si tratta degli argomenti da passare al programma. Anche in questo caso, per riconoscere l’ultimo elemento di questo array, gli si deve assegnare il puntatore nullo.

Le funzioni *execlp()* e *execvp()*, se ricevono solo il nome del file da eseguire, senza altre indicazioni del percorso, cercano questo file tra i percorsi indicati nella variabile di ambiente *PATH*, ammesso che sia dichiarata per il processo in corso.

Tutte le funzioni qui descritte, a eccezione di *execle()*, trasmettono al nuovo processo lo stesso ambiente (le stesse variabili di ambiente) del processo chiamante.

VALORE RESTITUITO

Queste funzioni, se hanno successo nel loro compito, non possono restituire alcunché, dato che in quel momento, il processo chiamante viene rimpiazzato da quello del file che viene eseguito. Pertanto, in caso contrario, queste funzioni possono re-

stituire soltanto un valore che rappresenta un errore, ovvero -1 , aggiornando anche la variabile *errno* di conseguenza.

ERRORI

Valore di <i>errno</i>	Significato
E2BIG	Ci sono troppi argomenti.
ENOMEM	Memoria insufficiente.
ENOENT	Il file richiesto non esiste.
EACCES	Il file non può essere avviato per la mancanza dei permessi di accesso necessari.
ENOEXEC	Il file non può essere un file eseguibile, perché non ne ha le caratteristiche.
EIO	Errore di input-output.

FILE SORGENTI

'lib/unistd.h' [[95.30](#)]

'lib/unistd/execl.c' [[95.30.10](#)]

'lib/unistd/execl.c' [[95.30.11](#)]

'lib/unistd/execlp.c' [[95.30.12](#)]

'lib/unistd/execv.c' [[95.30.13](#)]

'lib/unistd/execvp.c' [[95.30.15](#)]

'lib/unistd/execve.c' [[95.30.14](#)]

VEDERE ANCHE

execve(2) [[87.14](#)], *fork(2)* [[87.19](#)], *environ(7)* [[91.1](#)].

88.22 os32: `execl(3)`

Vedere *exec(3)* [[88.21](#)].

88.23 os32: `execle(3)`

Vedere *exec(3)* [[88.21](#)].

88.24 os32: `execlp(3)`

Vedere *exec(3)* [[88.21](#)].

88.25 os32: `execv(3)`

Vedere *exec(3)* [[88.21](#)].

88.26 os32: `execvp(3)`

Vedere *exec(3)* [[88.21](#)].

88.27 os32: `exit(3)`

Vedere *atexit(3)* [[88.7](#)].

88.28 os32: `fclose(3)`

NOME

‘**fclose**’ - chiusura di un flusso di file

SINTASSI

```
#include <stdio.h>
int fclose (FILE *fp);
```

DESCRIZIONE

La funzione *fclose()* chiude il flusso di file specificato tramite il puntatore *fp*. Questa realizzazione particolare di os32, si limita a richiamare la funzione *close()*, con l'indicazione del descrittore di file corrispondente al flusso.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
'EOF'	Errore: la variabile <i>errno</i> viene impostata di conseguenza.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore di file associato al flusso da chiudere, non è valido.

FILE SORGENTI

'lib/stdio.h' [95.18]

'lib/stdio/fclose.c' [95.18.3]

VEDERE ANCHE

close(2) [87.10], *fopen(3)* [88.36], *stdio(3)* [88.112].

88.29 os32: feof(3)

«

NOME

'feof' - verifica dello stato dell'indicatore di fine file

SINTASSI

```
#include <stdio.h>
int feof (FILE *fp);
```

DESCRIZIONE

La funzione *feof()* restituisce il valore dell'indicatore di fine file, riferito al flusso di file rappresentato da *fp*.

VALORE RESTITUITO

Valore	Significato
diverso da zero	Significa che l'indicatore di fine file è impostato.
0	Significa che l'indicatore di fine file non è impostato.

FILE SORGENTI

'lib/stdio.h' [[95.18](#)]

'lib/stdio/feof.c' [[95.18.4](#)]

VEDERE ANCHE

clearerr(3) [[88.12](#)], *ferror(3)* [[88.30](#)], *fileno(3)* [[88.35](#)], *stdio(3)* [[88.112](#)].

88.30 os32: *ferror(3)*



NOME

'**ferror**' - verifica dello stato dell'indicatore di errore

SINTASSI

```
#include <stdio.h>
int ferror (FILE *fp);
```

DESCRIZIONE

La funzione *ferror()* restituisce il valore dell'indicatore di errore, riferito al flusso di file rappresentato da *fp*.

VALORE RESTITUITO

Valore	Significato
diverso da zero	Significa che l'indicatore di errore è impostato.
0	Significa che l'indicatore di errore non è impostato.

FILE SORGENTI

'lib/stdio.h' [95.18]

'lib/stdio/ferror.c' [95.18.5]

VEDERE ANCHE

clearerr(3) [88.12], *feof(3)* [88.29], *fileno(3)* [88.35], *stdio(3)* [88.112].

88.31 os32: fflush(3)

«

NOME

'**fflush**' - fissaggio dei dati ancora sospesi nella memoria tampone

SINTASSI

```
#include <stdio.h>
int fflush (FILE *fp);
```


DESCRIZIONE

La funzione *fflush()* di *os32*, non fa alcunché, dato che non è prevista alcuna gestione della memoria tampone per i flussi di file.

VALORE RESTITUITO

Valore	Significato
0	Rappresenta il successo dell'operazione.

FILE SORGENTI

'lib/stdio.h' [95.18]

'lib/stdio/fflush.c' [95.18.6]

VEDERE ANCHE

fclose(3) [88.28], *fopen(3)* [88.36].

88.32 os32: fgetc(3)



NOME

'*fgetc*', '*getc*', '*getchar*' - lettura di un carattere da un flusso di file

SINTASSI

```
#include <stdio.h>

int fgetc (FILE *fp);
int getc (FILE *fp);
int getchar (void);
```

DESCRIZIONE

Le funzioni *fgetc()* e *getc()* sono equivalenti e leggono il carattere successivo dal flusso di file rappresentato da *fp*. La

funzione *getchar()* esegue la lettura di un carattere, ma dallo standard input.

VALORE RESTITUITO

In caso di successo, il carattere letto viene restituito in forma di intero positivo (il carattere viene inteso inizialmente senza segno, quindi viene trasformato in un intero, il quale rappresenta così un valore positivo). Se la lettura non può avere luogo, la funzione restituisce ‘**EOF**’, corrispondente a un valore negativo.

ERRORI

La variabile *errno* potrebbe risultare aggiornata nel caso la funzione restituisca ‘**EOF**’. Ma per saperlo, occorre azzerare la variabile *errno* prima della chiamata di *fgetc()*.

Valore di <i>errno</i>	Significato
EBADF	Il descrittore di file associato al flusso da cui leggere un carattere, non è valido.
EINVAL	Il descrittore di file associato al flusso da cui leggere un carattere, non consente un accesso in lettura.

FILE SORGENTI

‘lib/stdio.h’ [[95.18](#)]

‘lib/stdio/fgetc.c’ [[95.18.7](#)]

‘lib/stdio/getchar.c’ [[95.18.24](#)]

VEDERE ANCHE

fgets(3) [[88.34](#)], *gets(3)* [[88.34](#)].

88.33 os32: fgetpos(3)



NOME

‘**fgetpos**’, ‘**fsetpos**’ - lettura e impostazione della posizione corrente di un flusso di file

SINTASSI

```
#include <stdio.h>
int fgetpos (FILE *restrict fp, fpos_t *restrict pos);
int fsetpos (FILE *restrict fp, fpos_t *restrict pos);
```

DESCRIZIONE

Le funzioni *fgetpos()* e *fsetpos()*, rispettivamente, leggono o impostano la posizione corrente di un flusso di file.

Per os32, il tipo ‘**fpos_t**’ è dichiarato nel file ‘`stdio.h`’ come equivalente al tipo ‘**off_t**’ (file ‘`sys/types.h`’); tuttavia, la modifica di una variabile di tipo ‘**fpos_t**’ va fatta utilizzando una funzione apposita, perché lo standard consente che possa trattarsi anche di una variabile strutturata, i cui membri non sono noti.

L’uso della funzione *fgetpos()* comporta una modifica dell’informazione contenuta all’interno di **pos*, mentre la funzione *fsetpos()* usa il valore contenuto in **pos* per cambiare la posizione corrente del flusso di file. In pratica, si può usare *fsetpos()* solo dopo aver salvato una certa posizione con l’aiuto di *fgetpos()*.

Si comprende che il valore restituito dalle funzioni è solo un indice del successo o meno dell’operazione, dato che l’informazione sulla posizione viene ottenuta dalla modifica di una variabile di cui si fornisce il puntatore negli argomenti.

VALORE RESTITUITO

Valore	Significato
0	Rappresenta il successo dell'operazione.
-1	Indica il verificarsi di un errore, il quale può essere interpretato leggendo la variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore di file associato al flusso, non è valido.

FILE SORGENTI

'lib/stdio.h' [95.18]

'lib/stdio/fgetpos.c' [95.18.8]

'lib/stdio/fsetpos.c' [95.18.20]

VEDERE ANCHE

fseek(3) [88.44], *ftell(3)* [88.47], *rewind(3)* [88.100].

88.34 os32: fgets(3)

«

NOME

'**fgets**', '**gets**' - lettura di una stringa da un flusso di file

SINTASSI

```
#include <stdio.h>
char *fgets (char *restrict string, int n,
             FILE *restrict fp);
char *gets  (char *string);
```

DESCRIZIONE

La funzione *fgets()* legge una «riga» dal flusso di file *fp*, purché non più lunga di $n-1$ caratteri, collocandola a partire da ciò a cui punta *stringa*. La lettura termina al raggiungimento del carattere ‘\n’ (*new line*), oppure alla fine del file, oppure a $n-1$ caratteri. In ogni caso, viene aggiunto al termine, il codice nullo di terminazione di stringa: ‘\0’.

La funzione *gets()*, in modo analogo a *fgets()*, legge una riga dallo standard input, ma senza poter porre un limite massimo alla lunghezza della lettura.

VALORE RESTITUITO

In caso di successo, viene restituito il puntatore alla stringa contenente la riga letta, ovvero restituiscono *string*. In caso di errore, o comunque in caso di una lettura nulla, si ottiene ‘NULL’. La variabile *errno* viene aggiornata solo se si presenta un errore di accesso al file, mentre una lettura nulla, perché il flusso si è concluso, non comporta tale aggiornamento.

ERRORI

La variabile *errno* potrebbe risultare aggiornata nel caso le funzioni restituiscano ‘NULL’. Ma per saperlo, occorre azzerare la variabile *errno* prima della chiamata di queste.

Valore di <i>errno</i>	Significato
EBADF	Il descrittore di file associato al flusso da cui leggere un carattere, non è valido.
EINVAL	Il descrittore di file associato al flusso da cui leggere un carattere, non consente un accesso in lettura.

FILE SORGENTI

'lib/stdio.h' [95.18]

'lib/stdio/fgets.c' [95.18.9]

'lib/stdio/gets.c' [95.18.25]

VEDERE ANCHE

fgetc(3) [88.32], *getc(3)* [88.32].

88.35 os32: fileno(3)

«

NOME

'**fileno**' - traduzione di un flusso di file nel numero di descrittore corrispondente

SINTASSI

```
#include <stdio.h>
int fileno (FILE *fp);
```

DESCRIZIONE

La funzione *fileno()* traduce il flusso di file, rappresentato da *fp*, nel numero del descrittore corrispondente. Tuttavia, il risultato è valido solo se il flusso di file specificato è aperto effettivamente.

VALORE RESTITUITO

Se *fp* punta effettivamente a un flusso di file aperto, il valore restituito corrisponde al numero di descrittore del file stesso; diversamente, si potrebbe ottenere un numero privo di senso. Se come argomento si indica il puntatore nullo, si ottiene un errore, rappresentato dal valore -1.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	È stato richiesto di risolvere il puntatore nullo.

FILE SORGENTI

‘lib/stdio.h’ [95.18]

‘lib/stdio/FILE.c’ [95.18.1]

‘lib/stdio/fileno.c’ [95.18.10]

VEDERE ANCHE

clearerr(3) [88.12], *feof(3)* [88.29], *ferror(3)* [88.30], *stdio(3)* [88.112].

88.36 os32: fopen(3)

«

NOME

‘**fopen**’, ‘**freopen**’ - apertura di un flusso di file

SINTASSI

```
#include <stdio.h>
FILE *fopen    (const char *path, const char *mode);
FILE *freopen (const char *restrict path,
               const char *restrict mode,
               FILE *restrict fp);
```

DESCRIZIONE

La funzione *fopen()* apre il file indicato nella stringa a cui punta *path*, secondo le modalità di accesso contenute in *mode*, as-

sociandovi un flusso di file. In modo analogo agisce anche la funzione *freopen()*, la quale però, prima, chiude il flusso *fp*.

La modalità di accesso al file si specifica attraverso una stringa, come sintetizzato dalla tabella successiva.

<i>mode</i>	Significato
"r" "rb"	Si richiede un accesso in lettura, di un file già esistente. L'indice interno per l'accesso ai dati viene posizionato all'inizio del file.
"r+" "r+b" "rb+"	Si richiede un accesso in lettura e scrittura, di un file già esistente. L'indice interno per l'accesso ai dati viene posizionato all'inizio del file.
"w" "wb"	Si richiede un accesso in scrittura, di un file che viene troncato se esiste oppure viene creato. L'indice interno per l'accesso ai dati viene posizionato all'inizio del file.
"w+" "w+b" "wb+"	Si richiede un accesso in lettura e scrittura, di un file che viene troncato se esiste oppure viene creato. L'indice interno per l'accesso ai dati viene posizionato all'inizio del file.
"a" "ab"	Si richiede la creazione o il troncamento di un file, con accesso in aggiunta. L'indice interno per l'accesso ai dati viene posizionato alla fine del file, prima di ogni operazione di scrittura.
"a+" "a+b" "ab+"	Si richiede la creazione o il troncamento di un file, con accesso in lettura e scrittura. L'indice interno per l'accesso ai dati viene posizionato alla fine del file, prima di ogni operazione di scrittura.

VALORE RESTITUITO

Se l'operazione si conclude con successo, viene restituito il puntatore a ciò che rappresenta il flusso di file aperto. Se però si ot-

tiene un puntatore nullo (**‘NULL’**), si è verificato un errore che può essere interpretato dal contenuto della variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	È stato fornito un argomento non valido.
EPERM	Operazione non consentita.
EACCES	Accesso non consentito.
ENOENT	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
EROFS	Avendo richiesto un accesso in scrittura, si ottiene che il file system che lo contiene consente soltanto un accesso in lettura.
ENOTDIR	Il percorso che porta al file da aprire non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.
ENFILE	Non si possono aprire altri file nell'ambito del sistema operativo (il sistema ha raggiunto il limite).
EMFILE	Non si possono aprire altri file nell'ambito del processo in corso.

FILE SORGENTI

`'lib/stdio.h'` [[95.18](#)]

`'lib/stdio/FILE.c'` [[95.18.1](#)]

`'lib/stdio/fopen.c'` [[95.18.11](#)]

`'lib/stdio/freopen.c'` [[95.18.16](#)]

VEDERE ANCHE

open(2) [[87.37](#)], *fclose*(3) [[88.28](#)], *stdio*(3) [[88.112](#)].

88.37 os32: fprintf(3)

Vedere *printf(3)* [88.91].

88.38 os32: fputc(3)

NOME

‘**fputc**’, ‘**putc**’, ‘**putchar**’ - emissione di un carattere attraverso un flusso di file

SINTASSI

```
#include <stdio.h>
int fputc (int c, FILE *fp);
int putc (int c, FILE *fp);
int putchar (int c);
```

DESCRIZIONE

Le funzioni *fputc()* e *putc()* sono equivalenti e scrivono il carattere *c* nel flusso di file rappresentato da *fp*. La funzione *putchar()* esegue la scrittura di un carattere, ma nello standard output.

VALORE RESTITUITO

In caso di successo, il carattere scritto viene restituito in forma di intero positivo (il carattere viene inteso inizialmente senza segno, quindi viene trasformato in un intero, il quale rappresenta così un valore positivo). Se la scrittura non può avere luogo, la funzione restituisce ‘**EOF**’, corrispondente a un valore negativo.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore di file associato al flusso presso cui scrivere un carattere, non è valido.
EINVAL	Il descrittore di file associato al flusso presso cui scrivere un carattere, non consente un accesso in scrittura.

FILE SORGENTI

‘lib/stdio.h’ [95.18]

‘lib/stdio/fputc.c’ [95.18.13]

VEDERE ANCHE

fputs(3) [88.39], *puts(3)* [88.39].

88.39 os32: fputs(3)

«

NOME

‘**fputs**’, ‘**puts**’ - scrittura di una stringa attraverso un flusso di file

SINTASSI

```
#include <stdio.h>
int fputs (const char *restrict string, FILE *restrict fp);
int puts (const char *string);
```

DESCRIZIONE

La funzione *fputs()* scrive una stringa nel flusso di file *fp*, ma senza il carattere nullo di terminazione; la funzione *puts()* scrive

una stringa, aggiungendo anche il codice di terminazione ‘\n’, attraverso lo standard output.

VALORE RESTITUITO

Valore	Significato
≥ 0	Rappresenta il successo dell'operazione.
'EOF'	Indica il verificarsi di un errore, il quale può essere interpretato eventualmente leggendo la variabile <i>errno</i> .

ERRORI

La variabile *errno* potrebbe risultare aggiornata nel caso le funzioni restituiscano 'NULL'. Ma per saperlo, occorre azzerare la variabile *errno* prima della chiamata di queste.

Valore di <i>errno</i>	Significato
EBADF	Il descrittore di file associato al flusso verso cui si deve scrivere, non è valido.
EINVAL	Il descrittore di file associato al flusso verso cui si deve scrivere, non consente un accesso in scrittura.

FILE SORGENTI

'lib/stdio.h' [[95.18](#)]

'lib/stdio/fputs.c' [[95.18.14](#)]

'lib/stdio/puts.c' [[95.18.29](#)]

VEDERE ANCHE

fputc(3) [[88.38](#)], *putc(3)* [[88.38](#)], *putchar(3)* [[88.38](#)].

88.40 os32: fread(3)

**NOME**

‘**fread**’ - lettura di dati da un flusso di file

SINTASSI

```
#include <stdio.h>
size_t fread (void *restrict buffer, size_t size,
              size_t nmemb,
              FILE *restrict fp);
```

DESCRIZIONE

La funzione *fread()* legge *size*×*nmemb* byte dal flusso di file *fp*, trascrivendoli in memoria a partire dall’indirizzo a cui punta *buffer*.

VALORE RESTITUITO

La funzione restituisce la quantità di byte letta, diviso la dimensione del blocco *nmemb* (byte/*nmemb*). Se il valore ottenuto è inferiore a quello richiesto, occorre verificare eventualmente se ciò deriva dalla conclusione del file o da un errore, con l’aiuto di *feof(3)* [88.29] e di *ferror(3)* [88.30].

FILE SORGENTI

‘lib/stdio.h’ [95.18]

‘lib/stdio/fread.c’ [95.18.15]

VEDERE ANCHE

read(2) [87.39], *write(2)* [87.64], *feof(3)* [88.29], *ferror(3)* [88.30], *fwrite(3)* [88.49].

88.41 os32: free(3)

Vedere *malloc(3)* [88.76].

88.42 os32: freopen(3)

Vedere *fopen(3)* [88.36].

88.43 os32: fscanf(3)

Vedere *scanf(3)* [88.102].

88.44 os32: fseek(3)

NOME

‘**fseek**’, ‘**fseeko**’ - riposizionamento dell’indice di accesso di un flusso di file

SINTASSI

```
#include <stdio.h>
int fseek (FILE *fp, long int offset, int whence);
int fseeko (FILE *fp, off_t offset, int whence);
```

DESCRIZIONE

Le funzioni *fseek()* e *fseeko()* cambiano l’indice della posizione interna a un flusso di file, specificato dal parametro *fp*. L’indice viene collocato secondo lo scostamento rappresentato da *offset*, rispetto al riferimento costituito dal parametro *whence*. Il parametro *whence* può assumere solo tre valori, come descritto nello schema successivo.

Valore di <i>whence</i>	Significato
SEEK_SET	lo scostamento si riferisce all'inizio del file.
SEEK_CUR	lo scostamento si riferisce alla posizione che ha già l'indice interno al file.
SEEK_END	lo scostamento si riferisce alla fine del file.

La differenza tra le due funzioni sta solo nel tipo del parametro *offset*, il quale, da `'long int'` passa a `'off_t'`.

VALORE RESTITUITO

Valore	Significato
0	Rappresenta il successo dell'operazione.
-1	Indica il verificarsi di un errore, il quale può essere interpretato leggendo la variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore di file associato al flusso, non è valido.
EINVAL	Gli argomenti non sono validi, come succede se la combinazione di scostamento e riferimento non è ammissibile (per esempio uno scostamento negativo, quando il riferimento è l'inizio del file).

FILE SORGENTI

`'lib/stdio.h'` [[95.18](#)]

`'lib/stdio/FILE.c'` [[95.18.1](#)]

`'lib/stdio/fseek.c'` [[95.18.18](#)]

VEDERE ANCHE

lseek(2) [87.33], *fgetpos(3)* [88.33], *fsetpos(3)* [88.33], *ftell(3)* [88.47], *rewind(3)* [88.100].

88.45 os32: *fseeko(3)*

Vedere *fseek(3)* [88.44].

88.46 os32: *fsetpos(3)*

Vedere *fgetpos(3)* [88.33].

88.47 os32: *ftell(3)*

NOME

‘**ftell**’, ‘**ftello**’ - interrogazione dell’indice di accesso relativo a un flusso di file

SINTASSI

```
#include <stdio.h>
long int ftell (FILE *fp);
off_t ftello (FILE *fp);
```

DESCRIZIONE

Le funzioni *ftell()* e *ftello()* restituiscono il valore dell’indice interno di accesso al file specificato in forma di flusso, con il parametro *fp*. La differenza tra le due funzioni consiste nel tipo restituito, il quale, nel primo caso è ‘**long int**’, mentre nel secondo è ‘**off_t**’. L’indice ottenuto è riferito all’inizio del file.

VALORE RESTITUITO

Valore	Significato
≥ 0	Rappresenta l'indice interno al file.
-1	Indica il verificarsi di un errore, il quale può essere interpretato leggendo la variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore di file associato al flusso, non è valido.
EINVAL	Il flusso di file specificato non è valido.

FILE SORGENTI

'lib/stdio.h' [95.18]

'lib/stdio/FILE.c' [95.18.1]

'lib/stdio/ftell.c' [95.18.21]

'lib/stdio/ftello.c' [95.18.22]

VEDERE ANCHE

lseek(2) [87.33], *fgetpos*(3) [88.33], *fsetpos*(3) [88.33], *ftell*(3) [88.44], *rewind*(3) [88.100].

88.48 os32: *ftello*(3)

« Vedere *ftell*(3) [88.47].

88.49 os32: fwrite(3)



NOME

‘**fwrite**’ - scrittura attraverso un flusso di file

SINTASSI

```
#include <stdio.h>

size_t fwrite (const void *restrict buffer, size_t size,
               size_t nmemb,
               FILE *restrict fp);
```

DESCRIZIONE

La funzione *fwrite()* scrive *size*×*nmemb* byte nel flusso di file *fp*, traendoli dalla memoria, a partire dall’indirizzo a cui punta *buffer*.

VALORE RESTITUITO

La funzione restituisce la quantità di byte scritta, diviso la dimensione del blocco rappresentato da *nmemb* (byte/*nmemb*). Se il valore ottenuto è inferiore a quello richiesto, si tratta presumibilmente di un errore, ma per accertarsene conviene usare *ferror(3)* [88.30].

FILE SORGENTI

‘lib/stdio.h’ [95.18]

‘lib/stdio/fwrite.c’ [95.18.23]

VEDERE ANCHE

read(2) [87.39], *write(2)* [87.64], *feof(3)* [88.29], *ferror(3)* [88.30], *fread(3)* [88.40].

88.50 os32: `getc(3)`

« Vedere *fgetc(3)* [88.32].

88.51 os32: `getchar(3)`

« Vedere *fgetc(3)* [88.32].

88.52 os32: `getenv(3)`

«

NOME

‘**getenv**’ - lettura del valore di una variabile di ambiente

SINTASSI

```
#include <stdlib.h>
char *getenv (const char *name);
```

DESCRIZIONE

La funzione *getenv()* richiede come argomento una stringa contenente il nome di una variabile di ambiente, per poter restituire la stringa che rappresenta il contenuto di tale variabile.

VALORE RESTITUITO

Il puntatore alla stringa con il contenuto della variabile di ambiente richiesta, oppure il puntatore nullo (‘**NULL**’), se la variabile in questione non esiste.

FILE SORGENTI

‘lib/stdlib.h’ [95.19]

‘applic/crt0.mer.s’ [96.1.12]

‘`applic/crt0.sep.s`’ [96.1.13]

‘`lib/stdlib/environment.c`’ [95.19.8]

‘`lib/stdlib/getenv.c`’ [95.19.10]

VEDERE ANCHE

environ(7) [91.1], *putenv*(3) [88.94], *setenv*(3) [88.104],
unsetenv(3) [88.104].

88.53 os32: getgrent(3)



NOME

‘**getgrent**’, ‘**setgrent**’, ‘**endgrent**’ - accesso alle voci del file
‘`/etc/group`’

SINTASSI

```
#include <sys/types.h>
#include <grp.h>
struct group *getgrent (void);
void          setgrent (void);
void          endgrent (void);
```

DESCRIZIONE

La funzione *getgrent()* restituisce il puntatore a una variabile strutturata, di tipo ‘**struct group**’, come definito nel file ‘`grp.h`’, in cui si possono trovare le stesse informazioni contenute nelle voci (righe) del file ‘`/etc/group`’, separate in campi. La prima volta, nella variabile struttura a cui punta la funzione si ottiene il contenuto della prima voce, ovvero del primo utente dell’elenco; nelle chiamate successive si ottengono le altre.

Si utilizza la funzione *setgrent()* per ripartire dalla prima voce del file `/etc/group`; si utilizza invece la funzione *endgrent()* per chiudere il file `/etc/group` quando non serve più.

Il tipo `struct group` è definito nel file `grp.h` nel modo seguente:

```
struct group {
    char   *gr_name;
    char   *gr_passwd;
    gid_t  gr_gid;
    char   *gr_mem[];
};
```

La sequenza dei membri della struttura corrisponde a quella dei campi delle righe del file `/etc/group`; in particolare, l'ultimo membro raccoglie i riferimenti alle stringhe che descrivono i nomi degli utenti aggregati al gruppo.

VALORE RESTITUITO

La funzione *getgrent()* restituisce il puntatore a una variabile strutturata di tipo `struct group`, se l'operazione ha avuto successo. Se la scansione del file `/etc/group` ha raggiunto il termine, oppure se si è verificato un errore, restituisce invece il valore `NULL`. Per poter distinguere tra la conclusione del file o il verificarsi di un errore, prima della chiamata della funzione occorre azzerare il valore della variabile *errno*, verificando successivamente se ha acquisito un valore differente.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
EPERM	Operazione non consentita.
EACCES	Accesso non consentito.
ENOENT	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
ENOTDIR	Il percorso che porta al file da aprire non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.
ENFILE	Non si possono aprire altri file nell'ambito del sistema operativo (il sistema ha raggiunto il limite).
EMFILE	Non si possono aprire altri file nell'ambito del processo in corso.

DIFETTI

Nel file `/etc/group` non è possibile lasciare campi vuoti, a parte l'ultimo, anche se questi non sono usati, perché la scansione avviene con la funzione *strtok()*. Pertanto, benché il campo della parola d'ordine non sia usato con `os32`, occorre metterci ugualmente qualcosa.

FILE SORGENTI

`'lib/sys/types.h'` [[95.26](#)]

`'lib/pwd.h'` [[95.15](#)]

`'lib/grp/grent.c'` [[95.7.1](#)]

VEDERE ANCHE

getgrnam(3) [[88.54](#)], *getgrgid(3)* [[88.54](#)], *group(5)* [[90.1](#)].

88.54 os32: getgrnam(3)

<<

NOME

‘**getgrnam**’, ‘**getgrgid**’ - selezione di una voce dal file ‘/etc/passwd’

SINTASSI

```
#include <sys/types.h>
#include <grp.h>
struct group *getgrnam (const char *name);
struct group *getgrgid (gid_t gid);
```

DESCRIZIONE

La funzione *getgrnam()* restituisce il puntatore a una variabile strutturata, di tipo ‘**struct group**’, come definito nel file ‘grp.h’, contenente le informazioni sul gruppo specificato per nome, dal ‘/etc/group’. La funzione *getgrgid()* si comporta in modo analogo, individuando però il gruppo da selezionare in base al numero **GID**.

Il tipo ‘**struct group**’ è definito nel file ‘grp.h’ nel modo seguente:

```
struct group {
    char   *gr_name;
    char   *gr_passwd;
    gid_t  gr_gid;
    char   *gr_mem[];
};
```

La sequenza dei membri della struttura corrisponde a quella dei campi delle righe del file ‘/etc/group’; in particolare, l’ulti-

mo membro raccoglie i riferimenti alle stringhe che descrivono i nomi degli utenti aggregati al gruppo.

VALORE RESTITUITO

Le funzioni *getgrnam()* e *getgrgid()* restituiscono il puntatore a una variabile strutturata di tipo `'struct group'`, se l'operazione ha avuto successo. Se il nome o il numero del gruppo non si trovano nel file `'/etc/group'`, oppure se si presenta un errore, il valore restituito è `'NULL'`. Per poter distinguere tra una voce non trovata o il verificarsi di un errore di accesso al file `'/etc/group'`, prima della chiamata della funzione occorre azzerare il valore della variabile *errno*, verificando successivamente se ha acquisito un valore differente.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
EPERM	Operazione non consentita.
EACCES	Accesso non consentito.
ENOENT	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
ENOTDIR	Il percorso che porta al file da aprire non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.
ENFILE	Non si possono aprire altri file nell'ambito del sistema operativo (il sistema ha raggiunto il limite).
EMFILE	Non si possono aprire altri file nell'ambito del processo in corso.

FILE SORGENTI

'lib/sys/types.h' [95.26]

'lib/grp.h' [95.7]

'lib/grp/grent.c' [95.7.1]

VEDERE ANCHE

getgrent(3) [88.53], *setgrent(3)* [88.53], *endgrent(3)* [88.53],
group(5) [90.1].

88.55 os32: *getpwuid(3)*

«

Vedere *getgrnam(3)* [88.54].

88.56 os32: *getopt(3)*

«

NOME

'**getopt**' - scansione delle opzioni della riga di comando

SINTASSI

```
#include <unistd.h>
extern *char optarg;
extern int optind;
extern int opterr;
extern int optopt;
int getopt (int argc, char *const argv[],
            const char *optstring);
```

DESCRIZIONE

La funzione *getopt()* riceve, come primi due argomenti, gli stessi parametri *argc* e *argv*[], che sono già della funzione *main()* del

programma in cui *getopt()* si usa. In altri termini, *getopt()* deve conoscere la quantità degli argomenti usati per l'avvio del programma e deve poterli scandire. L'ultimo argomento di *getopt()* è una stringa contenente l'elenco delle lettere delle opzioni che ci si attende di trovare nella scansione delle stringhe dell'array *argv[]*, con altre sigle eventuali per sapere se tali opzioni sono singole o si attendono un proprio argomento.

Per poter usare la funzione *getopt()* proficuamente, è necessario che la sintassi di utilizzo del programma del quale si vuole scandire la riga di comando, sia uniforme con l'uso comune:

```
programma [-x [ argomento ] ] ... [ argomento ] ...
```

Pertanto, dopo il nome del programma possono esserci delle opzioni, riconoscibili perché composte da una sola lettera, preceduta da un trattino. Tali opzioni potrebbero richiedere un proprio argomento. Dopo le opzioni e i relativi argomenti, ci possono essere altri argomenti, al di fuori della competenza di *getopt()*. Vale anche la considerazione che più opzioni, prive di argomento, possono essere unite assieme in un'unica parola, con un solo trattino iniziale.

La funzione *getopt()* si avvale di variabili pubbliche, di cui occorre conoscere lo scopo.

La variabile *optind* viene usata da *getopt()* come indice per scandire l'array *argv[]*. Quando con gli utilizzi successivi di *optarg()* si determina che è stata completata la scansione delle opzioni (in quanto *optarg()* restituisce il valore -1), la variabile *optind* diventa utile per conoscere qual è il prossimo elemento di *argv[]*

da prendere in considerazione, trattandosi del primo argomento della riga di comando che non è un'opzione.

La variabile *opterr* serve per configurare il comportamento di *getopt()*. Questa variabile contiene inizialmente il valore 1. Quando *getopt()* incontra un'opzione per la quale si richiede un argomento, il quale risulta però mancante, se la variabile *opterr* risulta avere un valore diverso da zero, visualizza un messaggio di errore attraverso lo standard error. Pertanto, per evitare tale visualizzazione, è sufficiente assegnare preventivamente il valore zero alla variabile *opterr*.

Quando un'opzione individuata da *getopt()* risulta errata per qualche ragione (perché non prevista o perché si attende un argomento che invece non c'è), la variabile *optopt* riceve il valore (tradotto da carattere senza segno a intero) della lettera corrispondente a quell'opzione. Pertanto, in tal modo è possibile conoscere cosa ha provocato il problema.

Il puntatore *optarg* viene modificato quando *getopt()* incontra un'opzione che chiede un argomento. In tal caso, *optarg* viene modificato in modo da puntare alla stringa che rappresenta tale argomento.

La compilazione della stringa corrispondente a *optstring* deve avvenire secondo una sintassi precisa:

```
[:] [x [:]]...
```

La stringa *optstring* può iniziare con un simbolo di due punti, quindi seguono le lettere che rappresentano le opzioni possibili, tenendo conto che quelle per cui si attende un argomento devono

anche essere seguite da due punti. Per esempio, ‘**ab:cd:**’ significa che ci può essere un’opzione ‘**-a**’, un’opzione ‘**-b**’ seguita da un argomento, un’opzione ‘**-c**’ e un’opzione ‘**-d**’ seguita da un argomento.

Per comprendere l’uso della funzione *getopt()* si propone una versione ultraridotta di *kill(1)* [86.12], dove si ammette solo l’invio dei segnali SIGTERM e SIGQUIT.

```
#include <sys/os32.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <libgen.h>
//-----
int
main (int argc, char *argv[], char *envp[])
{
    int          signal = SIGTERM;
    int          pid;
    int          a;           // Index inside arguments.
    int          opt;
    extern char *optarg;
    extern int   optopt;
    //
    while ((opt = getopt (argc, argv, ":ls:")) != -1)
    {
        switch (opt)
```

```
{
  case 'l':
    printf ("TERM ");
    printf ("KILL ");
    printf ("\n");
    return (0);
    break;
  case 's':
    if (strcmp (optarg, "KILL") == 0)
      {
        signal = SIGKILL;
      }
    else if (strcmp (optarg, "TERM") == 0)
      {
        signal = SIGTERM;
      }
    break;
  case '?':
    fprintf (stderr, "Unknown option -%c.\n",
            optopt);
    return (1);
    break;
  case ':':
    fprintf (stderr, "Missing argument for option "
            "-%c\n", optopt);
    return (1);
    break;
  default:
    fprintf (stderr, "Getopt problem: unknown "
            "option %c\n", opt);
    return (1);
}
}
//
// Scan other command line arguments.
```

```
//  
for (a = optind; a < argc; a++)  
{  
    pid = atoi (argv[a]);  
    if (pid > 0)  
    {  
        if (kill (pid, signal) < 0)  
        {  
            perror (argv[a]);  
        }  
    }  
}  
return (0);  
}
```

Come si vede nell'esempio, la funzione *getopt()* viene chiamata sempre nello stesso modo, all'interno di un ciclo iterativo.

Alla prima chiamata della funzione, questa esamina il primo argomento della riga di comando, verificando se si tratta di un'opzione o meno. Se si tratta di un'opzione, benché possa essere errata per qualche ragione, restituisce un carattere (convertito a intero), il quale può corrispondere alla lettera dell'opzione se questa è valida, oppure a un simbolo differente in caso di problemi. Nelle chiamate successive, *getopt()* considera di volta in volta gli argomenti successivi della riga di comando, fino a quando si accorge che non ci sono più opzioni e restituisce semplicemente il valore -1 .

Durante la scansione delle opzioni, se *getopt()* restituisce il carattere '?', significa che ha incontrato un'opzione errata: potrebbe trattarsi di un'opzione non prevista, oppure di un'opzione che attende un argomento che non c'è. Tuttavia, la stringa *optstring*

potrebbe iniziare opportunamente con il simbolo di due punti, così come si vede nell'esempio. In tal caso, se *getopt()* incontra un'opzione errata in quanto mancante di un'opzione necessaria, invece di restituire '?', restituisce ':', così da poter distinguere il tipo di errore.

È il caso di osservare che le chiamate successive di *getopt()* fanno progredire la scansione della riga di comando e generalmente non c'è bisogno di tornare indietro per ripeterla. Tuttavia, nel caso lo si volesse, basterebbe reinizializzare la variabile *optind* a uno (il primo argomento della riga di comando).

FILE SORGENTI

'lib/unistd.h' [95.30]

'lib/unistd/getopt.c' [95.30.23]

88.57 os32: getpwent(3)

«

NOME

'getpwent', 'setpwent', 'endpwent' - accesso alle voci del file '/etc/passwd'

SINTASSI

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwent (void);
void          setpwent (void);
void          endpwent (void);
```


DESCRIZIONE

La funzione *getpwent()* restituisce il puntatore a una variabile strutturata, di tipo `'struct passwd'`, come definito nel file `'pwd.h'`, in cui si possono trovare le stesse informazioni contenute nelle voci (righe) del file `'/etc/passwd'`, separate in campi. La prima volta, nella variabile struttura a cui punta la funzione si ottiene il contenuto della prima voce, ovvero del primo utente dell'elenco; nelle chiamate successive si ottengono le altre.

Si utilizza la funzione *setpwent()* per ripartire dalla prima voce del file `'/etc/passwd'`; si utilizza invece la funzione *endpwent()* per chiudere il file `'/etc/passwd'` quando non serve più.

Il tipo `'struct passwd'` è definito nel file `'pwd.h'` nel modo seguente:

```
struct passwd {
    char    *pw_name;
    char    *pw_passwd;
    uid_t   pw_uid;
    gid_t   pw_gid;
    char    *pw_gecos;
    char    *pw_dir;
    char    *pw_shell;
};
```

La sequenza dei campi della struttura corrisponde a quella contenuta nel file `'/etc/passwd'`.

VALORE RESTITUITO

La funzione *getpwent()* restituisce il puntatore a una variabile strutturata di tipo `'struct passwd'`, se l'operazione ha avuto successo. Se la scansione del file `'/etc/passwd'` ha raggiunto

il termine, oppure se si è verificato un errore, restituisce invece il valore **'NULL'**. Per poter distinguere tra la conclusione del file o il verificarsi di un errore, prima della chiamata della funzione occorre azzerare il valore della variabile *errno*, verificando successivamente se ha acquisito un valore differente.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
EPERM	Operazione non consentita.
EACCES	Accesso non consentito.
ENOENT	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
ENOTDIR	Il percorso che porta al file da aprire non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.
ENFILE	Non si possono aprire altri file nell'ambito del sistema operativo (il sistema ha raggiunto il limite).
EMFILE	Non si possono aprire altri file nell'ambito del processo in corso.

DIFETTI

Nel file `'/etc/passwd'` non è possibile lasciare campi vuoti, anche se questi non sono usati, perché la scansione avviene con la funzione *strtok()*.

FILE SORGENTI

`'lib/sys/types.h'` [95.26]

`'lib/pwd.h'` [95.15]

‘lib/pwd/pwent.c’ [95.15.1]

VEDERE ANCHE

getpwnam(3) [88.58], *getpwuid(3)* [88.58], *passwd(5)* [90.4].

88.58 os32: getpwnam(3)



NOME

‘**getpwnam**’, ‘**getpwuid**’ - selezione di una voce dal file ‘/etc/passwd’

SINTASSI

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwnam (const char *name);
struct passwd *getpwuid (uid_t uid);
```

DESCRIZIONE

La funzione *getpwnam()* restituisce il puntatore a una variabile strutturata, di tipo ‘**struct passwd**’, come definito nel file ‘pwd.h’, contenente le informazioni sull’utenza specificata per nome, dal ‘/etc/passwd’. La funzione *getpwuid()* si comporta in modo analogo, individuando però l’utenza da selezionare in base al numero UID.

Il tipo ‘**struct passwd**’ è definito nel file ‘pwd.h’ nel modo seguente:

```
struct passwd {
    char    *pw_name;
    char    *pw_passwd;
    uid_t   pw_uid;
    gid_t   pw_gid;
    char    *pw_gecos;
    char    *pw_dir;
    char    *pw_shell;
};
```

La sequenza dei campi della struttura corrisponde a quella contenuta nel file `‘/etc/passwd’`.

VALORE RESTITUITO

Le funzioni *getpwnam()* e *getpwuid()* restituiscono il puntatore a una variabile strutturata di tipo `‘struct passwd’`, se l’operazione ha avuto successo. Se il nome o il numero dell’utente non si trovano nel file `‘/etc/passwd’`, oppure se si presenta un errore, il valore restituito è `‘NULL’`. Per poter distinguere tra una voce non trovata o il verificarsi di un errore di accesso al file `‘/etc/passwd’`, prima della chiamata della funzione occorre azzerare il valore della variabile *errno*, verificando successivamente se ha acquisito un valore differente.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
EPERM	Operazione non consentita.
EEXIST	Il file da creare esiste già.
EACCES	Accesso non consentito.
ENOENT	Il file non esiste, oppure non esiste il percorso che porta al file da creare.
EROFS	Avendo richiesto un accesso in scrittura, si ottiene che il file system che lo contiene consente soltanto un accesso in lettura.
ENOTDIR	Il percorso che porta al file da aprire non è valido, in quanto ciò che dovrebbe essere una directory, non lo è.
ENFILE	Non si possono aprire altri file nell'ambito del sistema operativo (il sistema ha raggiunto il limite).
EMFILE	Non si possono aprire altri file nell'ambito del processo in corso.

FILE SORGENTI

'lib/sys/types.h' [[95.26](#)]

'lib/pwd.h' [[95.15](#)]

'lib/pwd/pwent.c' [[95.15.1](#)]

VEDERE ANCHE

getpwent(3) [[88.57](#)], *setpwent(3)* [[88.57](#)], *endpwent(3)* [[88.57](#)], *passwd(5)* [[90.4](#)].

88.59 os32: getpwuid(3)

«

Vedere *getpwnam(3)* [88.58].

88.60 os32: gets(3)

«

Vedere *fgets(3)* [88.34].

88.61 os32: gmtime(3)

«

Vedere *ctime(3)* [88.15].

88.62 os32: htonl(3)

«

Vedere *byteorder(3)* [88.11].

88.63 os32: htons(3)

«

Vedere *byteorder(3)* [88.11].

88.64 os32: imaxabs(3)

«

Vedere *abs(3)* [88.3].

88.65 os32: imaxdiv(3)

«

Vedere *div(3)* [88.17].

88.66 os32: inet_ntop(3)

«

NOME

‘**inet_ntop**’ - conversione di un indirizzo IP dalla sua forma numerica a quella testuale

SINTASSI

```
#include <arpa/inet.h>

const char *inet_ntop (int family, const void *src,
                      char *dst, socklen_t size);
```

DESCRIZIONE

La funzione *inet_ntop()* converte un indirizzo IPv4, contenuto in una variabile strutturata di tipo ‘**struct in_addr**’, a cui punta *src*, in una stringa, annotandola nella memoria tampone a cui punta *dst*, la cui dimensione massima è di *size* byte.

Il parametro *family* indica il tipo di indirizzo da convertire, ma per os32 è ammesso soltanto il tipo *AF_INET*, corrispondente a IPv4.

VALORE RESTITUITO

Se l’operazione si conclude con successo, la funzione restituisce il puntatore a *dst*; altrimenti, si ottiene il valore *NULL* e va verificato il contenuto della variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EAFNOSUPPORT	È stata richiesta la conversione di un tipo di indirizzo diverso da <i>AF_INET</i> .
EINVAL	In corrispondenza del parametro <i>src</i> o di <i>dst</i> si ha un puntatore nullo.

FILE SORGENTI

‘lib/arpa/inet.h’ [95.3]

‘lib/arpa/inet/inet_ntop.c’ [95.3.3]

VEDERE ANCHE

inet_pton(3) [88.67].

88.67 os32: inet_pton(3)

«

NOME

‘**inet_pton**’ - conversione di un indirizzo IP dalla sua forma testuale a quella numerica

SINTASSI

```
#include <arpa/inet.h>
int inet_pton (int family, const char *src, void *dst);
```

DESCRIZIONE

La funzione *inet_pton()* converte un indirizzo IPv4, contenuto nella stringa *src*, in una variabile strutturata di tipo ‘**struct in_addr**’, a cui punta *dst*.

Il parametro *family* indica il tipo di indirizzo da convertire, ma per os32 è ammesso soltanto il tipo *AF_INET*, corrispondente a IPv4.

VALORE RESTITUITO

Se l’operazione si conclude con successo, la funzione restituisce il valore numerico 1; se invece la stringa *src* non contiene un indirizzo valido, la funzione restituisce 0; se il valore di *family* non è valido o non è gestito, restituisce -1.

ERRORI

La variabile *errno* non viene modificata, perché tutti i casi che possono presentarsi sono espressi attraverso il valore restituito dalla funzione.

FILE SORGENTI

‘lib/arpa/inet.h’ [95.3]

‘lib/arpa/inet/inet_pton.c’ [95.3.4]

VEDERE ANCHE

inet_ntop(3) [88.66].

88.68 os32: input_line(3)



NOME

‘input_line’ - riga di comando

SINTASSI

```
#include <sys/os32.h>
void input_line (char *line, char *prompt, size_t size,
                int type);
```

DESCRIZIONE

La funzione *input_line()* consente di inserire un’informazione da tastiera, interpretando in modo adeguato i codici usati per cancellare. Si tratta del mezzo principale di inserimento di un dato da tastiera per os32.

Il parametro *line* è il puntatore a un’area di memoria, da modificare con l’inserimento che si intende fare; questa area di memoria

deve essere in grado di contenere tanti byte quanto indicato con il parametro *size*. Il parametro *prompt* indica una stringa da usare come invito, a sinistra della riga da inserire. Il parametro *type* serve a specificare il tipo di visualizzazione sullo schermo di ciò che si inserisce. Si utilizzano della macro-variabili dichiarate nel file ‘`sys/os32.h`’:

Macro-variabile	Descrizione
<code>INPUT_LINE_ECHO</code>	Produce un comportamento «normale», per cui ciò che viene digitato è rappresentato conformemente sullo schermo del terminale.
<code>INPUT_LINE_HIDDEN</code>	Fa sì che quanto digitato non appaia: si usa per esempio per l’inserimento di una parola d’ordine.

La funzione conclude il suo funzionamento quando si preme [*Invio*].

VALORE RESTITUITO

La funzione non restituisce alcunché, ma ciò che viene digitato è disponibile nella memoria tampone rappresentata dal puntatore *line*, da intendere come stringa terminata correttamente.

FILE SORGENTI

‘`lib/sys/os32.h`’ [[95.21](#)]

‘`lib/sys/os32/input_line.c`’ [[95.21.1](#)]

VEDERE ANCHE

shell(1) [[86.24](#)].

login(1) [[86.14](#)].

88.69 os32: isatty(3)

**NOME**

‘**isatty**’ - verifica che un certo descrittore di file si riferisca a un terminale

SINTASSI

```
#include <unistd.h>
int isatty (int fdn);
```

DESCRIZIONE

La funzione *isatty()* verifica se il descrittore di file specificato con il parametro *fdn* si riferisce a un dispositivo di terminale.

VALORE RESTITUITO

Valore	Significato
1	Si tratta effettivamente del file di dispositivo di un terminale.
0	Il descrittore di file non riguarda un terminale, oppure si è verificato un errore; in ogni caso la variabile <i>errno</i> viene aggiornata.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il descrittore di file non si riferisce a un terminale.
EBADF	Il descrittore di file indicato non è valido.

FILE SORGENTI

‘lib/unistd.h’ [95.30]

‘lib/unistd/isatty.c’ [95.30.28]

VEDERE ANCHE

stat(2) [87.55], *ttynname(3)* [88.133].

88.70 os32: labs(3)

« Vedere *abs(3)* [88.3].

88.71 os32: ldiv(3)

« Vedere *div(3)* [88.17].

88.72 os32: llabs(3)

« Vedere *abs(3)* [88.3].

88.73 os32: lldiv(3)

« Vedere *div(3)* [88.17].

88.74 os32: major(3)

« Vedere *makedev(3)* [88.75].

88.75 os32: makedev(3)

«

NOME

'makedev', 'major', 'minor' - gestione dei numeri di dispositivo

SINTASSI

```
#include <sys/types.h>
dev_t makedev (int major, int minor);
int major (dev_t device);
int minor (dev_t device);
```

DESCRIZIONE

La funzione *makedev()* restituisce il numero di dispositivo complessivo, partendo dal numero primario (*major*) e dal numero secondario (*minor*), presi separatamente.

Le funzioni *major()* e *minor()*, rispettivamente, restituiscono il numero primario o il numero secondario, partendo da un numero di dispositivo completo.

Si tratta di funzioni non previste dallo standard, ma ugualmente diffuse.

FILE SORGENTI

'lib/sys/types.h' [95.26]

'lib/sys/types/makedev.c' [95.26.2]

'lib/sys/types/major.c' [95.26.1]

'lib/sys/types/minor.c' [95.26.3]

88.76 os32: malloc(3)



NOME

'**malloc**', '**free**', '**realloc**' - allocazione e rilascio dinamico di memoria

SINTASSI

```
#include <stdlib.h>
void *malloc (size_t size);
void free (void *address);
void *realloc (void *address, size_t size);
```

DESCRIZIONE

Le funzioni ‘...**alloc()**’ e *free()* consentono di allocare, riallocare e liberare delle aree di memoria, in modo dinamico.

La funzione *malloc()* (*memory allocation*) si usa per richiedere l’allocazione di una dimensione di almeno *size* byte di memoria. Se l’allocazione avviene con successo, la funzione restituisce il puntatore generico di tale area allocata.

Quando un’area di memoria allocata precedentemente non serve più, va liberata espressamente con l’ausilio della funzione *free()*, la quale richiede come argomento il puntatore generico all’inizio di tale area. Naturalmente, si può liberare la memoria una volta sola e un’area di memoria liberata non può più essere raggiunta.

Quando un’area di memoria già allocata richiede una modifica nella sua estensione, si può usare la funzione *realloc()*, la quale necessita di conoscere il puntatore precedente e la nuova estensione. La funzione restituisce un nuovo puntatore, il quale potrebbe eventualmente, ma non necessariamente, coincidere con quello dell’area originale.

Se le funzioni *malloc()* e *realloc()* falliscono nel loro intento, restituiscono un puntatore nullo.

VALORE RESTITUITO

Le funzioni *malloc()* e *realloc()* restituiscono il puntatore generico all’area di memoria allocata; se falliscono, restituiscono invece un puntatore nullo.

ERRORI

Valore di <i>errno</i>	Significato
ENOMEM	Memoria insufficiente.

FILE SORGENTI

‘lib/limits.h’ [95.1.6]

‘lib/stdlib.h’ [95.19]

‘lib/stdlib_alloc/malloc.c’ [95.19.24]

‘lib/stdlib_alloc/realloc.c’ [95.19.25]

‘lib/stdlib_alloc/free.c’ [95.19.23]

‘lib/stdlib_alloc/_alloc_list.c’ [95.19.22]

88.77 os32: memccpy(3)

«

NOME

‘**memccpy**’ - copia di un’area di memoria

SINTASSI

```
#include <string.h>
void *memccpy (void *restrict dst, const void *restrict org, int
               size_t n);
```

DESCRIZIONE

La funzione *memccpy()* copia al massimo *n* byte a partire dall’area di memoria a cui punta *org*, verso l’area che inizia da *dst*, fermandosi se si incontra il carattere *c*, il quale viene copiato regolarmente, fermo restando il limite massimo di *n* byte.

Le due aree di memoria, origine e destinazione, non devono sovrapporsi.

VALORE RESTITUITO

Nel caso in cui la copia sia avvenuta con successo, fino a incontrare il carattere *c*, la funzione restituisce il puntatore al carattere successivo a *c*, nell'area di memoria di destinazione. Se invece tale carattere non viene trovato nei primi *n* byte, restituisce il puntatore nullo 'NULL'. La variabile *errno* non viene modificata.

FILE SORGENTI

'lib/string.h' [95.20]

'lib/string/memccpy.c' [95.20.1]

VEDERE ANCHE

memcpy(3) [88.80], *memmove(3)* [88.81], *strcpy(3)* [88.117], *strncpy(3)* [88.117].

88.78 os32: memchr(3)

«

NOME

'**memchr**' - scansione della memoria alla ricerca di un carattere

SINTASSI

```
#include <string.h>
void *memchr (const void *memory, int c, size_t n);
```

DESCRIZIONE

La funzione *memchr()* scandisce l'area di memoria a cui punta *memory*, fino a un massimo di *n* byte, alla ricerca del carattere *c*.

VALORE RESTITUITO

Se la funzione trova il carattere, restituisce il puntatore al carattere trovato, altrimenti restituisce il puntatore nullo **'NULL'**.

FILE SORGENTI

'lib/string.h' [95.20]

'lib/string/memchr.c' [95.20.2]

VEDERE ANCHE

strchr(3) [88.114], *strrchr(3)* [88.114], *strpbrk(3)* [88.125].

88.79 os32: memcmp(3)



NOME

'memcmp' - confronto di due aree di memoria

SINTASSI

```
#include <string.h>
int memcmp (const void *memory1, const void *memory2,
           size_t n);
```

DESCRIZIONE

La funzione *memcmp()* confronta i primi *n* byte di memoria delle aree che partono, rispettivamente, da *memory1* e da *memory2*.

VALORE RESTITUITO

Valore	Esito del confronto.
-1	<i>memory1</i> < <i>memory2</i>
0	<i>memory1</i> == <i>memory2</i>
+1	<i>memory1</i> > <i>memory2</i>

FILE SORGENTI

'lib/string.h' [95.20]

'lib/string/memcmp.c' [95.20.3]

VEDERE ANCHE

strcmp(3) [88.115], *strncmp(3)* [88.115].

88.80 os32: memcpy(3)

<<

NOME

'**memcpy**' - copia di un'area di memoria

SINTASSI

```
#include <string.h>
void *memcpy (void *restrict dst, const void *restrict org,
              size_t n);
```

DESCRIZIONE

La funzione *memcpy()* copia al massimo *n* byte a partire dall'area di memoria a cui punta *org*, verso l'area che inizia da *dst*.

Le due aree di memoria, origine e destinazione, non devono sovrapporsi.

VALORE RESTITUITO

La funzione restituisce *dst*.

FILE SORGENTI

'lib/string.h' [95.20]

'lib/string/memcpy.c' [95.20.4]

VEDERE ANCHE

memccpy(3) [88.77], *memmove(3)* [88.81], *strcpy(3)* [88.117], *strncpy(3)* [88.117].

88.81 os32: memmove(3)



NOME

‘**memmove**’ - copia di un’area di memoria

SINTASSI

```
#include <string.h>
void *memmove (void *dst, const void *org, size_t n);
```

DESCRIZIONE

La funzione *memmove()* copia al massimo *n* byte a partire dall’area di memoria a cui punta *org*, verso l’area che inizia da *dst*. A differenza di quanto fa *memcpy()*, la funzione *memmove()* esegue la copia correttamente anche se le due aree di memoria sono sovrapposte.

VALORE RESTITUITO

La funzione restituisce *dst*.

FILE SORGENTI

‘lib/string.h’ [95.20]

‘lib/string/memmove.c’ [95.20.5]

VEDERE ANCHE

memccpy(3) [88.77], *memcpy(3)* [88.80], *strcpy(3)* [88.117], *strncpy(3)* [88.117].

88.82 os32: memset(3)

<<

NOME

‘**memset**’ - scrittura della memoria con un byte sempre uguale

SINTASSI

```
#include <string.h>
void *memset (void *memory, int c, size_t n);
```

DESCRIZIONE

La funzione *memset()* scrive *n* byte, contenenti il valore di *c*, ridotto a un carattere, a partire dal ciò a cui punta *memory*.

FILE SORGENTI

‘lib/string.h’ [95.20]

‘lib/string/memset.c’ [95.20.6]

VEDERE ANCHE

memcpy(3) [88.80].

88.83 os32: minor(3)

<<

Vedere *makedev(3)* [88.75].

88.84 os32: mktime(3)

<<

Vedere *ctime(3)* [88.15].

88.85 os32: namep(3)



NOME

‘**namep**’ - ricerca del percorso di un programma utilizzando la variabile di ambiente **PATH**

SINTASSI

```
#include <sys/os32.h>
int namep (const char *name, char *path, size_t size);
```

DESCRIZIONE

La funzione **namep()** trova il percorso di un programma, tenendo conto delle informazioni contenute nella variabile di ambiente **PATH**.

Il parametro **name** rappresenta una stringa con il nome del comando da cercare nel file system; il parametro **path** deve essere il puntatore di un’area di memoria, da sovrascrivere con il percorso assoluto del programma da avviare, una volta trovato, con l’accortezza di far sì che risulti una stringa terminata correttamente; il parametro **size** specifica la dimensione massima che può avere la stringa **path**.

Questa funzione viene utilizzata in particolare da **execvp()**.

VALORE RESTITUITO

Valore	Significato
0	Operazione riuscita.
-1	Operazione fallita. Va verificato l’errore indicato dalla variabile errno .

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Uno degli argomenti non è valido.
ENOENT	La variabile di ambiente <i>PATH</i> non è dichiarata, oppure il comando richiesto non si trova nei percorsi previsti.
ENAMETOOLONG	Il percorso per l'avvio del programma è troppo lungo.

FILE SORGENTI

'lib/sys/os32.h' [95.21]

'lib/sys/os32/namep.c' [95.21.4]

VEDERE ANCHE

shell(1) [86.24], *execvp(3)* [88.21], *execlp(3)* [88.21].

88.86 os32: ntohl(3)

«

Vedere *byteorder(3)* [88.11].

88.87 os32: ntohs(3)

«

Vedere *byteorder(3)* [88.11].

88.88 os32: offsetof(3)

«

NOME

'**offsetof**' - posizione di un membro di una struttura, dall'inizio della stessa

SINTASSI

```
#include <stddef.h>
size_t offsetof (type, member);
```

DESCRIZIONE

La macroistruzione *offsetof()* consente di determinare la collocazione relativa di un membro di una variabile strutturata, restituendo la quantità di byte che la struttura occupa prima dell'inizio del membro richiesto. Per ottenere questo risultato, il primo argomento deve essere il nome del tipo del membro cercato, mentre il secondo argomento deve essere il nome del membro stesso.

VALORE RESTITUITO

La macroistruzione restituisce lo scostamento del membro specificato, rispetto all'inizio della struttura a cui appartiene, espresso in byte.

FILE SORGENTI

'lib/stddef.h' [[95.1.12](#)]

88.89 os32: opendir(3)

NOME

'**opendir**' - apertura di una directory

SINTASSI

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (const char *name);
```

DESCRIZIONE

La funzione *opendir()* apre la directory rappresentata da *name*, posizionando l'indice interno per le operazioni di accesso alla prima voce della directory stessa.

VALORE RESTITUITO

La funzione restituisce il puntatore al flusso aperto; in caso di errore, restituisce 'NULL' e aggiorna la variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il valore di <i>name</i> non è valido.
EMFILE	Troppi file aperti dal processo.
ENFILE	Troppi file aperti complessivamente nel sistema.
ENOTDIR	Il percorso indicato in <i>name</i> non corrisponde a una directory.

NOTE

La funzione *opendir()* attiva il bit *close-on-exec*, rappresentato dalla macro-variabile *FD_CLOEXEC*, per il descrittore del file che rappresenta la directory. Ciò serve a garantire che la directory venga chiusa quando si utilizzano le funzioni 'exec... ()'.

FILE SORGENTI

'lib/sys/types.h' [95.26]

'lib/dirent.h' [95.4]

'lib/dirent/DIR.c' [95.4.1]

'lib/dirent/opendir.c' [95.4.3]

VEDERE ANCHE

open(2) [87.37], *closedir(3)* [88.13], *readdir(3)* [88.98],
rewinddir(3) [88.101].

88.90 os32: perror(3)

NOME

‘**perror**’ - emissione di un messaggio di errore di sistema

SINTASSI

```
#include <stdio.h>
void perror (const char *string);
```

DESCRIZIONE

La funzione *perror()* legge il valore della variabile *errno* e, se questo è diverso da zero, emette attraverso lo standard output la stringa fornita come argomento, ammesso che non si tratti del puntatore nullo, quindi continua con l'emissione della descrizione dell'errore.

La funzione *perror()* di os32, nell'emettere il testo dell'errore, mostra anche il nome del file sorgente e il numero della riga in cui si è verificato. Ma questi dati sono validi soltanto se l'annotazione dell'errore è avvenuta, a suo tempo, con l'ausilio della funzione *errset(3)* [88.20], la quale non è prevista dagli standard.

FILE SORGENTI

‘lib/errno.h’ [95.5]

‘lib/stdio.h’ [95.18]

‘lib/stdio/perror.c’ [95.18.26]

VEDERE ANCHE

errno(3) [88.20], *strerror(3)* [88.120].

88.91 os32: printf(3)

«

NOME

‘**printf**’, ‘**fprintf**’, ‘**sprintf**’, ‘**snprintf**’ - composizione dei dati per la visualizzazione

SINTASSI

```
#include <stdio.h>
int printf (char *restrict format, ...);
int fprintf (FILE *fp, char *restrict format, ...);
int sprintf (char *restrict string,
             const char *restrict format, ...);
int snprintf (char *restrict string, size_t size,
             const char *restrict format, ...);
```

DESCRIZIONE

Le funzioni del gruppo ‘**...printf()**’ hanno in comune lo scopo di comporre dei dati in forma di stringa, generalmente per la visualizzazione, o comunque per la fruizione a livello umano.

I dati in ingresso possono essere vari e si collocano come argomenti finali, di tipo e quantità non noti nel prototipo delle funzioni. Per quantificare e qualificare questi argomenti aggiuntivi, la stringa a cui punta il parametro *format*, deve contenere degli *specificatori di conversione*, oltre eventualmente ad altri caratteri. Pertanto, queste funzioni, prendono la stringa a cui punta

format, la interpretano e determinano quali argomenti variabili sono presenti, quindi producono un'altra stringa, composta dalla stringa precedente, sostituendo gli specificatori di conversione con i dati a cui questi si riferiscono, secondo una forma di conversione definita dagli specificatori stessi. Si osservi l'esempio seguente:

```
printf ("Valore: %x %i %o\n", 123, 124, 125);
```

In questo modo si ottiene la visualizzazione, attraverso lo standard output, della stringa **Valore: 7b 124 175**. Infatti: **'%x'** è uno specificatore di conversione che richiede di interpretare il proprio parametro (in questo caso il primo) come intero normale e di rappresentarlo in esadecimale; **'%i'** legge un numero intero normale e lo rappresenta nella forma decimale consueta; **'%o'** legge un intero e lo mostra in ottale.

La funzione *printf()* emette il risultato della composizione attraverso lo standard output; la funzione *fprintf()* lo fa attraverso il flusso di file *fp*; le funzioni *sprintf()* e *snprintf()* si limitano a scrivere il risultato a partire da ciò a cui punta *string*, con la particolarità di *snprintf()* che si dà comunque un limite da non superare, per evitare che la scrittura vada a sovrascrivere altri dati in memoria.

Gli specificatori di conversione devono rispettare la sintassi seguente per la libreria di `os32`:

```
% [ simbolo ] [ n_ampiezza ] [ .n_precision ] [ hh | h | l | ll | j | z | t ] tipo
```

La prima cosa da individuare in uno specificatore di conversione è il tipo di argomento che viene interpretato e, di conseguenza, il

genere di rappresentazione che se ne vuole produrre. Il tipo viene espresso da una lettera alfabetica, alla fine dello specificatore di conversione.

Simbolo	Tipo di argomento	Conversione applicata
%...d %...i	int	Numero intero con segno da rappresentare in base dieci.
%...u	unsigned int	Numero intero senza segno da rappresentare in base dieci.
%...o	unsigned int	Numero intero senza segno da rappresentare in ottale (senza lo zero iniziale che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
%...x %...X	unsigned int	Numero intero senza segno da rappresentare in esadecimale (senza il prefisso '0x' o '0X' che viene usato spesso per caratterizzare un tale tipo di rappresentazione).
%...c	int	Un carattere singolo, dopo la conversione in ' unsigned char '.
%...b	unsigned int	Numero intero senza segno da rappresentare in binario (si tratta di un'estensione non prevista nello standard).
%...s	char *	Una stringa.
%%		Questo specificatore si limita a produrre un carattere di percentuale ('%') che altrimenti non sarebbe rappresentabile.

Nel modello sintattico che descrive lo specificatore di conversio-

ne, si vede che subito dopo il segno di percentuale può apparire un simbolo (*flag*).

Simbolo	Corrispondenza
%+... %+0 <i>ampiezza</i> ...	Il segno «+» fa sì che i numeri con segno lo mostrino anche se è positivo. Può combinarsi con lo zero.
%0 <i>ampiezza</i> ... %+0 <i>ampiezza</i> ...	Lo zero fa sì che siano inseriti degli zeri a sinistra per allineare a destra il valore, nell'ambito dell'ampiezza specificata. Può combinarsi con il segno «+».
% <i>ampiezza</i> ... % <i> </i> <i>ampiezza</i> ...	In mancanza di uno zero iniziale, in presenza dell'indicazione dell'ampiezza, il valore viene allineato a destra usando degli spazi. È possibile esprimere esplicitamente l'intenzione di usare gli spazi mettendo proprio uno spazio, ma in generale non è richiesto. Se si mette lo spazio letteralmente, questo non è poi compatibile con lo zero, mentre le combinazioni con gli altri simboli sono ammissibili.
%- <i>ampiezza</i> ... %-+ <i>ampiezza</i> ...	Il segno meno, usato quando la conversione prevede l'uso di una quantità fissa di caratteri con un valore che appare di norma allineato a destra, fa sì che il risultato sia allineato a sinistra. Il segno meno si può combinare il segno «+» e il cancelletto.

Subito prima della lettera che definisce il tipo di conversione, possono apparire una o due lettere che modificano la lunghezza del valore da interpretare (per lunghezza si intende qui la quantità di byte usati per rappresentarlo). Per esempio, '%...li' indica che la conversione riguarda un valore di tipo 'long int'. Tra questi specificatori della lunghezza del dato in ingresso ce ne sono

alcuni che indicano un rango inferiore a quello di **'int'**, come per esempio **'%...hhd'** che si riferisce a un numero intero della dimensione di un **'signed char'**; in questi casi occorre comunque considerare che nella trasmissione degli argomenti alle funzioni interviene sempre la promozione a intero, pertanto viene letto il dato della dimensione specificata, ma viene «consumato» il risultato ottenuto dalla promozione.

Simbolo	Tipo	Simbolo	Tipo
%...hhd %...hhi	signed char	%...hhu %...hho %...hhx %...hhX %...hbb	unsigned char
%...hd %...hi	short int	%...hu %...ho %...hx %...hX %...hb	unsigned short int

Simbolo	Tipo	Simbolo	Tipo
%...ld %...li	long int	%...lu %...lo %...lx %...lX %...lb	unsigned long int
%...lld %...lli	long long int	%...llu %...llo %...llx %...lLX %...llb	unsigned long long int
Simbolo	Tipo	Simbolo	Tipo
%...jd %...ji	intmax_t	%...ju %...jo %...jx %...jX %...jb	uintmax_t

Simbolo	Tipo	Simbolo	Tipo
%...zd %...zi	size_t	%...zu %...zo %...zx %...zX %...zb	size_t

Simbolo	Tipo	Simbolo	Tipo
%...td %...ti	ptrdiff_t	%...tu %...to %...tx %...tX %...tb	ptrdiff_t

Tra il simbolo (*flag*) e il modificatore di lunghezza può apparire un numero che rappresenta l'ampiezza da usare nella trasformazione ed eventualmente la precisione: '*ampiezza* [*.precisione*]' . Per os32, la precisione si applica esclusivamente alle stringhe, la quale specifica la quantità di caratteri da considerare, troncando il resto.

VALORE RESTITUITO

Le funzioni restituiscono la quantità di caratteri utilizzati nella composizione della nuova stringa, escluso il carattere nullo di terminazione.

FILE SORGENTI

'lib/stdio.h' [95.18]

'lib/stdio/FILE.c' [95.18.1]

'lib/stdio/fprintf.c' [95.18.12]

'lib/stdio/printf.c' [95.18.27]

'lib/stdio/sprintf.c' [95.18.35]

'lib/stdio/snprintf.c' [95.18.34]

VEDERE ANCHE

vfprintf(3) [88.137], *vprintf(3)* [88.137], *vsprintf(3)* [88.137],
vsnprintf(3) [88.137], *scanf(3)* [88.102].

88.92 os32: *putc(3)*

Vedere *fputc(3)* [88.38].

«

88.93 os32: *putchar(3)*

Vedere *fputc(3)* [88.38].

«

88.94 os32: *putenv(3)*

«

NOME

'**putenv**' - assegnamento di una variabile di ambiente

SINTASSI

```
#include <stdlib.h>
int putenv (const char *string);
```

DESCRIZIONE

La funzione `putenv()` assegna una variabile di ambiente. Se questa esiste già, va a rimpiazzare il valore assegnatole in precedenza, altrimenti la crea contestualmente.

La funzione richiede un solo parametro, costituito da una stringa in cui va specificato il nome della variabile e il contenuto da assegnargli, usando la forma `'nome=valore'`. Per esempio, `'PATH=/bin:/usr/bin'`.

VALORE RESTITUITO

Valore	Significato
0	Operazione riuscita.
-1	Operazione fallita. Va verificato l'errore indicato dalla variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Argomento non valido.
ENOMEM	Memoria insufficiente.

FILE SORGENTI

`'lib/stdlib.h'` [95.19]

`'applic/crt0.mer.s'` [96.1.12]

`'applic/crt0.sep.s'` [96.1.13]

`'lib/stdlib/environment.c'` [95.19.8]

`'lib/stdlib/putenv.c'` [95.19.15]

VEDERE ANCHE

environ(7) [91.1], *getenv*(3) [88.52], *setenv*(3) [88.104],
unsetenv(3) [88.104].

88.95 os32: *puts*(3)

Vedere *fputs*(3) [88.39].

88.96 os32: *qsort*(3)

NOME

‘**qsort**’ - riordino di un array

SINTASSI

```
#include <stdlib.h>
void qsort (void *base, size_t nmemb, size_t size,
            int (*compare)(const void *, const void *));
```

DESCRIZIONE

La funzione *qsort*() riordina un array composto da *nmemb* elementi da *size* byte ognuno. Il primo argomento, ovvero il parametro *base*, è il puntatore all’indirizzo iniziale di questo array in memoria.

Il riordino avviene comparando i vari elementi con l’ausilio di una funzione, passata tramite il suo puntatore, la quale deve ricevere due argomenti, costituiti dai puntatori agli elementi dell’array da confrontare. Tale funzione deve restituire un valore minore di zero per un confronto in cui il suo primo argomento deve essere collocato prima del secondo; un valore pari a zero se

gli argomenti sono uguali ai fini del riordino; un valore maggiore di zero se il suo primo argomento va collocato dopo il secondo nel riordino.

Segue un esempio di utilizzo della funzione *qsort()*:

```
#include <stdio.h>
#include <stdlib.h>

int confronta (const void *a, const void *b)
{
    int x = *((int *) a);
    int y = *((int *) b);
    return x - y;
}

int main (void)
{
    int a[] = {3, 1, 5, 2};

    qsort (&a[0], 4, sizeof (int), confronta);
    printf ("%d %d %d %d\n", a[0], a[1], a[2], a[3]);

    return 0;
}
```

FILE SORGENTI

‘lib/stdlib.h’ [95.19]

‘lib/stdlib/qsort.c’ [95.19.16]

88.97 os32: rand(3)

**NOME**

‘**rand**’ - generazione di numeri pseudo-casuali

SINTASSI

```
#include <stdlib.h>
int  rand  (void);
void srand (unsigned int seed);
```

DESCRIZIONE

La funzione *rand()* produce un numero intero casuale, sulla base di un seme, il quale può essere cambiato in ogni momento, con l’ausilio di *srand()*. A ogni chiamata della funzione *rand()*, il risultato ottenuto, viene utilizzato anche come seme per la chiamata successiva. Se inizialmente non viene assegnato alcun seme, il primo valore predefinito è pari a 1.

VALORE RESTITUITO

La funzione *rand()* restituisce un numero intero casuale, determinato sulla base del seme accumulato in precedenza.

FILE SORGENTI

‘lib/stdlib.h’ [[95.19](#)]

‘lib/stdlib/rand.c’ [[95.19.17](#)]

88.98 os32: readdir(3)

**NOME**

‘**readdir**’ - lettura di una directory

SINTASSI

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir (DIR *dp);
```

DESCRIZIONE

La funzione *readdir()* legge una voce dalla directory rappresentata da *dp* e restituisce il puntatore a una variabile strutturata di tipo ‘**struct dirent**’, contenente le informazioni tratte dalla voce letta. La variabile strutturata in questione si trova in memoria statica e viene sovrascritta con le chiamate successive della funzione *readdir()*.

Il tipo ‘**struct dirent**’ è definito nel file di intestazione ‘*dirent.h*’, nel modo seguente:

```
struct dirent {
    ino_t      d_ino;
    char      d_name[NAME_MAX+1];
};
```

Il membro *d_ino* è il numero di inode del file il cui nome appare nel membro *d_name*. La macro-variabile *NAME_MAX* è dichiarata a sua volta nel file di intestazione ‘*limits.h*’. La dimensione del membro *d_name* è tale da permettere di includere anche il valore zero di terminazione delle stringhe.

VALORE RESTITUITO

La funzione restituisce il puntatore a una variabile strutturata di tipo `'struct dirent'`; se la lettura ha già raggiunto la fine della directory, oppure per qualunque altro tipo di errore, la funzione restituisce `'NULL'` e aggiorna eventualmente la variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	La directory rappresentata da <i>dp</i> non è valida.

FILE SORGENTI

`'lib/sys/types.h'` [95.26]

`'lib/dirent.h'` [95.4]

`'lib/dirent/DIR.c'` [95.4.1]

`'lib/dirent/readdir.c'` [95.4.4]

VEDERE ANCHE

read(2) [87.39], *closedir(3)* [88.13], *opendir(3)* [88.89], *rewinddir(3)* [88.101].

88.99 os32: *realloc(3)*

Vedere *malloc(3)* [88.76].

88.100 os32: *rewind(3)*

NOME

`'rewind'` - riposizionamento all'inizio dell'indice di accesso a un flusso di file

SINTASSI

```
#include <stdio.h>
void rewind (FILE *fp);
```

DESCRIZIONE

La funzione *rewind()* azzera l'indice della posizione interna del flusso di file specificato con il parametro *fp*; inoltre azzera anche l'indicatore di errore dello stesso flusso. In pratica si ottiene la stessa cosa di:

```
(void) fseek (fp, 0L, SEEK_SET);
clearerr (fp);
```

FILE SORGENTI

'lib/stdio.h' [95.18]

'lib/stdio/FILE.c' [95.18.1]

'lib/stdio/rewind.c' [95.18.30]

VEDERE ANCHE

lseek(2) [87.33], *fgetpos(3)* [88.33], *fsetpos(3)* [88.33], *ftell(3)* [88.47], *fseek(3)* [88.44], *rewind(3)* [88.100].

88.101 os32: *rewinddir(3)*

«

NOME

'**rewinddir**' - riposizionamento all'inizio del riferimento per l'accesso a una directory

SINTASSI

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir (DIR *dp);
```

DESCRIZIONE

La funzione *rewinddir()* riposiziona i riferimenti per l'accesso alla directory indicata, in modo che la prossima lettura o scrittura avvenga dalla prima posizione.

VALORE RESTITUITO

La funzione non restituisce alcunché e non si presenta nemmeno la possibilità di segnalare errori attraverso la variabile *errno*.

FILE SORGENTI

'lib/sys/types.h' [95.26]

'lib/dirent.h' [95.4]

'lib/dirent/DIR.c' [95.4.1]

'lib/dirent/rewinddir.c' [95.4.5]

VEDERE ANCHE

rewind(3) [88.100], *closedir(3)* [88.13], *opendir(3)* [88.89], *rewinddir(3)* [88.98].

88.102 os32: scanf(3)



NOME

'**scanf**', '**fscanf**', '**sscanf**' - interpretazione dell'input e conversione

SINTASSI

```
#include <stdio.h>

int scanf (const char *restrict format, ...);

int fscanf (FILE *restrict fp,
            const char *restrict format, ...);

int sscanf (char *restrict string,
            const char *restrict format, ...);
```

DESCRIZIONE

Le funzioni del gruppo ‘...**scanf()**’ hanno in comune lo scopo di interpretare dei dati, forniti in forma di stringa, convertendoli opportunamente.

I dati in ingresso sono costituiti da una sequenza di caratteri, la quale viene fornita tramite lo standard input per *scanf()*, tramite il flusso di file *fp* per *fscanf()*, oppure tramite la stringa *string* per *sscanf()*. L’interpretazione dei dati in ingresso viene guidata da una stringa di formato, costituita dal parametro *format*, per le tre funzioni. La stringa di formato contiene degli *specificatori di conversione*, con cui si determina il tipo degli argomenti variabili che non sono esplicitati nel prototipo delle funzioni.

Per ogni specificatore di conversione contenuto nella stringa di formato, deve esistere un argomento, successivo al parametro *format*, costituito dal puntatore a una variabile di tipo conforme a quanto indicato dallo specificatore relativo. La conversione per quello specificatore, comporta la memorizzazione del risultato in memoria, in corrispondenza del puntatore relativo. Si osservi l’esempio seguente:

```
int valore;  
...  
scanf ("%i", &valore);
```

In questo modo si attende l’inserimento, attraverso lo standard input, di un numero intero, da convertire e assegnare così alla variabile *valore*; Infatti, lo specificatore di conversione ‘%i’, consente di interpretare un numero intero.

Gli specificatori di conversione devono rispettare la sintassi seguente per la libreria di os32:

```
% [*] [n_ampiezza] [hh | h | l | j | z | t] tipo
```

Come si può vedere, all’inizio può apparire un asterisco, il cui scopo è quello di annullare l’assegnamento del valore a una variabile. In pratica, con l’asterisco il dato corrispondente allo specificatore viene interpretato, ma poi non viene salvato.

Successivamente può apparire un numero che rappresenta l’ampiezza del dato da interpretare, in byte, il cui scopo è quello di limitare la lettura fino a un certo carattere.

Dopo può apparire una sigla, composta da una o più lettere, il cui scopo è quello di modificare la dimensione predefinita della variabile di destinazione. In altri termini, senza questo modificatore si intende che la variabile ricevente debba essere di una certa grandezza, ma con l’aggiunta del «modificatore di lunghezza» si precisa invece qualcosa di diverso. In pratica, il modificatore di lunghezza usato da queste funzioni è equivalente a quello delle funzioni di composizione dell’output.

Al termine dello specificatore di conversione appare una lette-

ra che dichiara come deve essere interpretato il dato in ingresso e, in mancanza del modificatore di lunghezza, indica anche la dimensione della variabile ricevente.

Tipi di conversione.

Simbolo	Tipo di argomento	Conversione applicata
<code>%...d</code>	<code>int *</code>	Numero intero con segno rappresentato in base dieci.
<code>%...i</code>	<code>int *</code>	Numero intero con segno rappresentare in base dieci o in base otto, avendo come prefisso uno zero, oppure in base sedici, avendo come prefisso <code>'0x'</code> o <code>'0X'</code> .
<code>%...u</code>	<code>unsigned int *</code>	Numero intero senza segno rappresentato in base dieci.
<code>%...o</code>	<code>unsigned int *</code>	Numero intero senza segno rappresentato in ottale (con o senza lo zero iniziale).

Simbolo	Tipo di argomento	Conversione applicata
<code>%...x</code>	<code>unsigned int *</code>	Numero intero senza segno rappresentato in esadecimale (con o senza il prefisso <code>'0x'</code> o <code>'0X'</code>).
<code>%...c</code>	<code>char *</code>	Interpreta un solo carattere, o più caratteri se si specifica l'ampiezza. Nella lettura contano anche gli spazi o qualunque altro carattere e non viene aggiunto il carattere nullo di terminazione.
<code>%...s</code>	<code>char *</code>	Interpreta una sequenza di caratteri che non siano spazi, aggiungendo alla fine il carattere nullo di terminazione.
<code>%p</code>	<code>void *</code>	Interpreta il valore di un puntatore che sia rappresentato nello stesso modo in cui farebbe la funzione <code>'printf("%p", puntatore)'</code> .

Simbolo	Tipo di argomento	Conversione applicata
<code>%n</code>	<code>int *</code>	Questo specificatore non esegue alcuna conversione e si limita a memorizzare la quantità di caratteri ('char') letti fino a quel punto.
<code>%... [...]</code>	<code>char *</code>	Interpreta una stringa non vuota contenente solo i caratteri elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri si cerca anche la parentesi quadra chiusa, questa va messa all'inizio dell'elenco: '%... [...]' .
<code>%... [^...]</code>	<code>char *</code>	Interpreta una stringa non vuota contenente solo caratteri diversi da quelli elencati tra parentesi quadre, aggiungendo alla fine il carattere nullo di terminazione. Se tra i caratteri da escludere si vuole indicare anche la parentesi quadra chiusa, questa va messa all'inizio dell'elenco: '%... [^] ...' .
<code>%%</code>		Interpreta un carattere di percentuale tra i dati in ingresso, ma senza memorizzare alcunché.

Modificatori della lunghezza del dato in uscita.

Simbolo	Tipo	Simbolo	Tipo
%...hhd %...hhi	signed char *	%...hhu %...hho %...hhx %...hhn	unsigned char *
%...hd %...hi	short int *	%...hu %...ho %...hx %...hn	unsigned short int *
%...ld %...li	long int *	%...lu %...lo %...lx %...ln	unsigned long int *

Simbolo	Tipo	Simbolo	Tipo
%...jd %...ji	intmax_t *	%...ju %...jo %...jx %...jn	uintmax_t *
%...zd %...zi	size_t *	%...zu %...zo %...zx %...zn	size_t *
%...td %...ti	ptrdiff_t *	%...tu %...to %...tx %...tn	ptrdiff_t *

La stringa di conversione è composta da *direttive*, ognuna delle quali è formata da: uno o più spazi (spazi veri e propri o caratteri di tabulazione orizzontale); un carattere diverso da ‘%’ e diverso dai caratteri che rappresentano spazi, oppure uno specificatore di conversione.

[*spazi*] *carattere* | %...

Dalla sequenza di caratteri che costituisce i dati in ingresso da

interpretare, vengono eliminati automaticamente gli spazi iniziali e finali (tutto ciò che si può considerare spazio, anche il codice di interruzione di riga), quando all'inizio o alla fine non ci sono corrispondenze con specificatori di conversione che possono interpretarli.

Quando la direttiva di interpretazione inizia con uno o più spazi orizzontali, significa che si vogliono ignorare gli spazi a partire dalla posizione corrente nella lettura dei dati in ingresso; inoltre, la presenza di un carattere che non fa parte di uno specificatore di conversione indica che quello stesso carattere deve essere incontrato nell'interpretazione dei dati in ingresso, altrimenti il procedimento di lettura e valutazione si deve interrompere. Se due specificatori di conversione appaiono adiacenti, i dati in ingresso corrispondenti possono essere separati da spazi orizzontali o da spazi verticali (il codice di interruzione di riga).

VALORE RESTITUITO

Le funzioni restituiscono la quantità di elementi in ingresso interpretati e assegnati correttamente: una quantità inferiore al previsto indica pertanto un errore. Se le funzioni restituiscono il valore **'EOF'**, si tratta di un errore, dovuto eventualmente a un problema di interpretazione del formato o a un problema di accesso al flusso di file da cui deve provenire l'input.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
EPERM	Operazione di accesso non consentita.
EACCES	Accesso non consentito.
EBADF	Il descrittore del file a cui si riferisce il flusso, non è valido.
ERANGE	Il risultato della conversione di un intero, non può essere memorizzato nel tipo di variabile a cui si riferisce lo specificatore di conversione.

FILE SORGENTI

‘lib/stdio.h’ [95.18]

‘lib/stdio/fscanf.c’ [95.18.17]

‘lib/stdio/scanf.c’ [95.18.31]

‘lib/stdio/sscanf.c’ [95.18.36]

VEDERE ANCHE

vfscanf(3) [88.138], *vscanf(3)* [88.138], *vsscanf(3)* [88.138], *printf(3)* [88.91].

88.103 os32: *setbuf(3)*

«

NOME

‘**setbuf**’, ‘**setvbuf**’ - modifica della memoria tampone per i flussi di file

SINTASSI

```
#include <stdio.h>
void setbuf (FILE *restrict fp, char *restrict buffer);
int setvbuf (FILE *restrict fp, char *restrict buffer,
            int buf_mode, size_t size);
```

DESCRIZIONE

Le funzioni *setbuf()* e *setvbuf()* della libreria di os32, non fanno alcunché, perché os32 non gestisce una memoria tampone per i flussi di file.

VALORE RESTITUITO

La funzione *setvbuf()* restituisce, in tutti i casi, il valore zero.

FILE SORGENTI

‘lib/stdio.h’ [95.18]

‘lib/stdio/setbuf.c’ [95.18.32]

‘lib/stdio/setvbuf.c’ [95.18.33]

VEDERE ANCHE

fflush(3) [88.31].

88.104 os32: setenv(3)



NOME

‘**setenv**’, ‘**unsetenv**’ - assegnamento o cancellazione di una variabile di ambiente

SINTASSI

```
#include <stdlib.h>

int setenv (const char *name, const char *value,
           int overwrite);

int unsetenv (const char *name);
```

DESCRIZIONE

La funzione *setenv()* crea o assegna un valore a una variabile di ambiente. Se questa variabile esiste già, la modifica del valore assegnatole può avvenire soltanto se l'argomento corrispondente al parametro *overwrite* risulta essere diverso da zero; in caso contrario, la modifica non ha luogo.

La funzione *unsetenv()* si limita a cancellare la variabile di ambiente specificata come argomento.

VALORE RESTITUITO

Valore	Significato
0	Operazione riuscita.
-1	Operazione fallita. Va verificato l'errore indicato dalla variabile <i>errno</i> .

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Argomento non valido.
ENOMEM	Memoria insufficiente.

FILE SORGENTI

'lib/stdlib.h' [[95.19](#)]

`'applic/crt0.mer.s'` [[96.1.12](#)]

`'applic/crt0.sep.s'` [[96.1.13](#)]

`'lib/stdlib/environment.c'` [[95.19.8](#)]

`'lib/stdlib/setenv.c'` [[95.19.18](#)]

`'lib/stdlib/unsetenv.c'` [[95.19.21](#)]

VEDERE ANCHE

environ(7) [[91.1](#)], *getenv*(3) [[88.52](#)], *putenv*(3) [[88.94](#)].

88.105 os32: *setgrent*(3)

Vedere *getgrent*(3) [[88.53](#)].

«

88.106 os32: *setpwent*(3)

Vedere *getpwent*(3) [[88.57](#)].

«

88.107 os32: *setvbuf*(3)

Vedere *setbuf*(3) [[88.103](#)].

«

88.108 os32: *snprintf*(3)

Vedere *printf*(3) [[88.91](#)].

«

88.109 os32: *sprintf*(3)

Vedere *printf*(3) [[88.91](#)].

«

88.110 os32: srand(3)

«
Vedere *rand(3)* [88.97].

88.111 os32: sscanf(3)

«
Vedere *scanf(3)* [88.102].

88.112 os32: stdio(3)

«

NOME

‘**stdio**’ - libreria per la gestione dei file in forma di flussi di file (*stream*)

SINTASSI

```
#include <stdio.h>
```

DESCRIZIONE

Le funzioni di libreria che fanno capo al file di intestazione ‘`stdio.h`’, consentono di gestire i file in forma di «flussi», rappresentati da puntatori al tipo ‘**FILE**’. Questa gestione si sovrappone a quella dei file in forma di «descrittori», la quale avviene tramite chiamate di sistema. Lo scopo della sovrapposizione dovrebbe essere quello di gestire i file con l’ausilio di una memoria tampone, cosa che però la libreria di os32 non fornisce. Nella libreria di os32, il tipo ‘**FILE ***’ è un puntatore a una variabile strutturata che contiene solo tre informazioni: il numero del descrittore del file a cui il flusso si associa; lo stato di errore; lo stato di raggiungimento della fine del file.

```
typedef struct {
    int         fdn;        // File descriptor number.
    char        error;     // Error indicator.
    char        eof;       // End of file indicator.
} FILE;
```

Le variabili strutturate necessarie per questa gestione, sono raccolte in un array, dichiarato nel file `lib/stdio/FILE.c`, con il nome `_stream[]`, dove per il descrittore di file *n*, si associano sempre i dati di `_stream[n]`.

```
FILE _stream[FOPEN_MAX];
```

Così come sono previsti tre descrittori (zero, uno e due) per la gestione di standard input, standard output e standard error, tutti i processi inizializzano l'array `_stream[]` con l'abbinamento a tali descrittori, per i primi tre flussi.

```
void
_stdio_stream_setup (void)
{
    _stream[0].fdn    = 0;
    _stream[0].error = 0;
    _stream[0].eof    = 0;

    _stream[1].fdn    = 1;
    _stream[1].error = 0;
    _stream[1].eof    = 0;

    _stream[2].fdn    = 2;
    _stream[2].error = 0;
    _stream[2].eof    = 0;
}
```

Ciò avviene attraverso il codice contenuto nel file `crt0.s`, dove si chiama la funzione che provvede a tale inizializzazione, conte-

nuta nel file `'lib/stdio/FILE.c'`. Per fare riferimento ai flussi predefiniti, si usano i nomi `'stdin'`, `'stdout'` e `'stderr'`, i quali sono dichiarati nel file `'stdio.h'`, come puntatori ai primi tre elementi dell'array `_stream[]`:

```
#define stdin    (&_stream[0])
#define stdout   (&_stream[1])
#define stderr   (&_stream[2])
```

FILE SORGENTI

`'lib/sys/types.h'` [95.26]

`'lib/stdio.h'` [95.18]

`'lib/stdio/FILE.c'` [95.18.1]

`'applic/crt0.mer.s'` [96.1.12]

`'applic/crt0.sep.s'` [96.1.13]

VEDERE ANCHE

`close(2)` [87.10], `open(2)` [87.37], `read(2)` [87.39], `write(2)` [87.64].

88.113 os32: strcat(3)

«

NOME

`'strcat'`, `'strncat'` - concatenamento di una stringa a un'altra già esistente

SINTASSI

```
#include <string.h>
char *strcat (char *restrict dst,
             const char *restrict org);
char *strncat (char *restrict dst,
              const char *restrict org,
              size_t n);
```

DESCRIZIONE

Le funzioni *strcat()* e *strncat()* copiano la stringa di origine *org*, aggiungendola alla stringa di destinazione *dst*, nel senso che la scrittura avviene a partire dal codice di terminazione ‘\0’ che viene così sovrascritto. Al termine della copia, viene aggiunto nuovamente il codice di terminazione di stringa ‘\0’, nella nuova posizione conclusiva.

Nel caso particolare di *strncat()*, la copia si arresta al massimo dopo il trasferimento di *n* caratteri. Pertanto, la stringa di origine per *strncat()* potrebbe anche non essere terminata correttamente, se raggiunge o supera la dimensione di *n* caratteri. In ogni caso, nella destinazione viene aggiunto il codice nullo di terminazione di stringa, dopo la copia del carattere *n*-esimo.

VALORE RESTITUITO

Le due funzioni restituiscono *dst*.

FILE SORGENTI

‘lib/string.h’ [95.20]

‘lib/string/strcat.c’ [95.20.7]

‘lib/string/strncat.c’ [95.20.16]

VEDERE ANCHE

memccpy(3) [88.77], *memcpy(3)* [88.80], *strcpy(3)* [88.117], *strncpy(3)* [88.117].

88.114 os32: strchr(3)

<<

NOME

‘**strchr**’, ‘**strrchr**’ - ricerca di un carattere all’interno di una stringa

SINTASSI

```
#include <string.h>
char *strchr (const char *string, int c);
char *strrchr (const char *string, int c);
```

DESCRIZIONE

Le funzioni *strchr()* e *strrchr()* scandiscono la stringa *string* alla ricerca di un carattere uguale al valore di *c*. La funzione *strchr()* scandisce a partire da «sinistra», ovvero ricerca la prima corrispondenza con il carattere *c*, mentre la funzione *strrchr()* cerca l’ultima corrispondenza con il carattere *c*, pertanto è come se scandisse da «destra».

VALORE RESTITUITO

Se le due funzioni trovano il carattere che cercano, ne restituiscono il puntatore, altrimenti restituiscono ‘**NULL**’.

FILE SORGENTI

‘lib/string.h’ [95.20]

'lib/string/strchr.c' [95.20.8]

'lib/string/strrchr.c' [95.20.20]

VEDERE ANCHE

memchr(3) [88.78], *strlen(3)* [88.121], *strpbrk(3)* [88.125], *strspn(3)* [88.127].

88.115 os32: strcmp(3)



NOME

'**strcmp**', '**strncmp**' - confronto di due stringhe

SINTASSI

```
#include <string.h>
int strcmp (const char *string1, const char *string2);
int strncmp (const char *string1, const char *string2,
             size_t n);
int strcoll (const char *string1, const char *string2);
```

DESCRIZIONE

Le funzioni *strcmp()* e *strncmp()* confrontano due stringhe, nel secondo caso, il confronto avviene al massimo fino al *n*-esimo carattere.

La funzione *strcoll()* dovrebbe eseguire il confronto delle due stringhe tenendo in considerazione la configurazione locale. Tuttavia, os32 non è in grado di gestire le configurazioni locali, pertanto questa funzione coincide esattamente con *strcmp()*.

VALORE RESTITUITO

Valore	Esito del confronto.
-1	<i>string1</i> < <i>string2</i>
0	<i>string1</i> == <i>string2</i>
+1	<i>string1</i> > <i>string2</i>

FILE SORGENTI

'lib/string.h' [[95.20](#)]

'lib/string/strcmp.c' [[95.20.9](#)]

'lib/string/strncmp.c' [[95.20.17](#)]

'lib/string/strcoll.c' [[95.20.10](#)]

VEDERE ANCHE

memcmp(3) [[88.79](#)].

88.116 os32: *strcoll(3)*

<<

Vedere *strcmp(3)* [[88.115](#)].

88.117 os32: *strcpy(3)*

<<

NOME

'*strcpy*', '*strncpy*' - copia di una stringa

SINTASSI

```
#include <string.h>
char *strcpy (char *restrict dst,
              const char *restrict org);
char *strncpy (char *restrict dst,
              const char *restrict org,
              size_t n);
```

DESCRIZIONE

Le funzioni *strcpy()* e *strncpy()*, copiano la stringa *org*, completa di codice nullo di terminazione, nella destinazione *dst*. Eventualmente, nel caso di *strncpy()*, la copia non supera i primi *n* caratteri, con l'aggravante che in tal caso, se nei primi *n* caratteri non c'è il codice nullo di terminazione delle stringhe, nella destinazione *dst* si ottiene una stringa non terminata.

VALORE RESTITUITO

Le funzioni restituiscono *dst*.

FILE SORGENTI

'lib/string.h' [95.20]

'lib/string/strcpy.c' [95.20.11]

'lib/string/strncpy.c' [95.20.18]

VEDERE ANCHE

memccpy(3) [88.77], *memcpy(3)* [88.80], *memmove(3)* [88.81].

88.118 os32: *strcspn(3)*

Vedere *strspn(3)* [88.127].

88.119 os32: *strdup(3)*

NOME

'*strdup*' - duplicazione di una stringa

SINTASSI

```
#include <string.h>
char *strdup (const char *string);
```

DESCRIZIONE

La funzione *strdup()*, alloca dinamicamente una quantità di memoria, necessaria a copiare la stringa *string*, quindi esegue tale copia e restituisce il puntatore alla nuova stringa allocata. Tale puntatore può essere usato successivamente per liberare la memoria, con l'ausilio della funzione *free()*.

VALORE RESTITUITO

La funzione restituisce il puntatore alla nuova stringa ottenuta dalla copia, oppure **NULL** nel caso non fosse possibile allocare la memoria necessaria.

ERRORI

Valore di <i>errno</i>	Significato
ENOMEM	Memoria insufficiente.

FILE SORGENTI

'lib/string.h' [95.20]

'lib/string/strdup.c' [95.20.13]

VEDERE ANCHE

free(3) [88.76], *malloc(3)* [88.76], *realloc(3)* [88.76].

88.120 os32: *strerror(3)*

«

NOME

'**strerror**' - descrizione di un errore in forma di stringa

SINTASSI

```
#include <string.h>
char *strerror (int errnum);
```

DESCRIZIONE

La funzione *strerror()* interpreta il valore *errnum* come un errore, di quelli che può rappresentare la variabile *errno* del file ‘*errno.h*’.

VALORE RESTITUITO

La funzione restituisce il puntatore a una stringa contenente la descrizione dell’errore, oppure soltanto ‘**Unknown error**’, se l’argomento ricevuto non è traducibile.

FILE SORGENTI

‘*lib/errno.h*’ [[95.5](#)]

‘*lib/string.h*’ [[95.20](#)]

‘*lib/string/strerror.c*’ [[95.20.14](#)]

VEDERE ANCHE

errno(3) [[88.20](#)], *perror(3)* [[88.90](#)].

88.121 os32: *strlen(3)*

NOME

‘**strlen**’ - lunghezza di una stringa

SINTASSI

```
#include <string.h>
size_t strlen (const char *string);
```

DESCRIZIONE

La funzione *strlen()* calcola la lunghezza della stringa, ovvero la quantità di caratteri che la compone, escludendo il codice nullo di conclusione.

VALORE RESTITUITO

La funzione restituisce la quantità di caratteri che compone la stringa, escludendo il codice ‘\0’ finale.

FILE SORGENTI

‘lib/string.h’ [95.20]

‘lib/string/strlen.c’ [95.20.15]

88.122 os32: strncat(3)

« Vedere *strcat(3)* [88.113].

88.123 os32: strncmp(3)

« Vedere *strcmp(3)* [88.115].

88.124 os32: strncpy(3)

« Vedere *strcpy(3)* [88.117].

88.125 os32: strpbrk(3)

«

NOME

‘**strpbrk**’ - scansione di una stringa alla ricerca di un carattere

SINTASSI

```
#include <string.h>
char *strpbrk (const char *string, const char *accept);
```

DESCRIZIONE

La funzione *strpbrk()* cerca il primo carattere, nella stringa *string*, che corrisponda a uno di quelli contenuti nella stringa *accept*.

VALORE RESTITUITO

Restituisce il puntatore al primo carattere che, nella stringa *string* corrisponde a uno di quelli contenuti nella stringa *accept*. In mancanza di alcuna corrispondenza, restituisce 'NULL'.

FILE SORGENTI

'lib/string.h' [95.20]

'lib/string/strpbrk.c' [95.20.19]

VEDERE ANCHE

memchr(3) [88.78], *strchr(3)* [88.114], *strstr(3)* [88.128], *strtok(3)* [88.129].

88.126 os32: *strrchr(3)*

Vedere *strchr(3)* [88.114].



88.127 os32: strspn(3)

**NOME**

‘**strspn**’, ‘**strcspn**’ - scansione di una stringa, limitatamente a un certo insieme di caratteri

SINTASSI

```
#include <string.h>
size_t strspn (const char *string, const char *accept);
size_t strcspn (const char *string, const char *reject);
```

DESCRIZIONE

La funzione *strspn()* scandisce la stringa *string*, calcolando la lunghezza di questa che contiene, a partire dall’inizio, soltanto caratteri che si trovano nella stringa *accept*.

La funzione *strcspn()* scandisce la stringa *string*, calcolando la lunghezza di questa che contiene, a partire dall’inizio, soltanto caratteri che non si trovano nella stringa *reject*.

VALORE RESTITUITO

La funzione *strspn()* restituisce la lunghezza della stringa che contiene soltanto caratteri contenuti in *accept*.

La funzione *strcspn()* restituisce la lunghezza della stringa che contiene soltanto caratteri che non sono contenuti in *reject*.

FILE SORGENTI

‘lib/string.h’ [95.20]

‘lib/string/strspn.c’ [95.20.21]

‘lib/string/strcspn.c’ [95.20.12]

VEDERE ANCHE

memchr(3) [88.78], *strchr(3)* [88.114], *strpbrk(3)* [88.125], *strstr(3)* [88.128], *strtok(3)* [88.129].

88.128 os32: strstr(3)



NOME

‘**strstr**’ - ricerca di una sottostringa

SINTASSI

```
#include <string.h>
char *strstr (const char *string, const char *substring);
```

DESCRIZIONE

La funzione *strstr()* scandisce la stringa *string*, alla ricerca della prima corrispondenza con la stringa *substring*, restituendo eventualmente il puntatore all’inizio di tale corrispondenza.

VALORE RESTITUITO

Se la ricerca termina con successo, viene restituito il puntatore all’inizio della sottostringa contenuta in *string*; diversamente viene restituito il puntatore nullo ‘**NULL**’.

FILE SORGENTI

‘lib/string.h’ [95.20]

‘lib/string/strstr.c’ [95.20.22]

VEDERE ANCHE

memchr(3) [88.78], *strchr(3)* [88.114], *strpbrk(3)* [88.125], *strtok(3)* [88.129].

88.129 os32: strtok(3)

**NOME**

‘**strtok**’ - *string token*, ovvero estrazione di pezzi da una stringa

SINTASSI

```
#include <string.h>
char *strtok (char *restrict string,
              const char *restrict delim);
```

DESCRIZIONE

La funzione *strtok()* serve a suddividere una stringa in unità, definite *token*, specificando un elenco di caratteri da intendere come delimitatori, in una seconda stringa. La funzione va usata in fasi successive, fornendo solo inizialmente la stringa da suddividere che continua poi a essere utilizzata se al suo posto viene fornito il puntatore nullo. La funzione restituisce, di volta in volta, il puntatore alla sottostringa contenente l'unità individuata, oppure il puntatore nullo, se non può trovarla.

La funzione deve tenere memoria di un puntatore in modo persistente e deve isolare le unità modificando la stringa originale, inserendo il carattere nullo di terminazione alla fine delle unità individuate.

Quando la funzione viene chiamata indicando al posto della stringa da scandire il puntatore nullo, l'insieme dei delimitatori può essere diverso da quello usato nelle fasi precedenti.

Per comprendere lo scopo della funzione viene utilizzato lo stesso esempio che appare nel documento *ISO/IEC 9899:TC2*, al para-

grafo 7.21.5.7, con qualche piccola modifica per poterlo rendere un programma autonomo:

```
#include <stdio.h>
#include <string.h>
int
main (void)
{
    char str[] = "?a???b,,,#c";
    char *t;

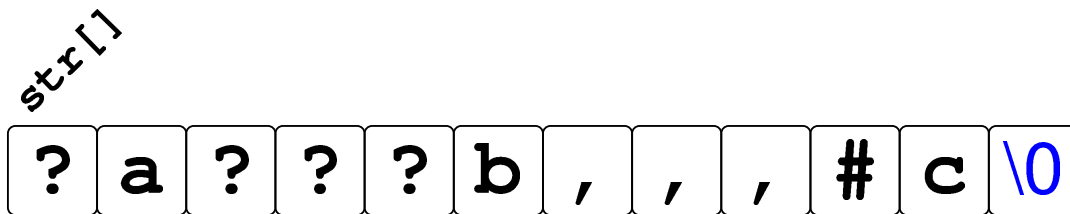
    t = strtok (str, "?");           // t punta all'unità "a"
    printf ("strtok: \"%s\"\n", t);
    t = strtok (NULL, ",");         // t punta all'unità "??b"
    printf ("strtok: \"%s\"\n", t);
    t = strtok (NULL, "#,");        // t punta all'unità "c"
    printf ("strtok: \"%s\"\n", t);
    t = strtok (NULL, "?");         // t è un puntatore nullo
    printf ("strtok: \"%s\"\n", t);

    return 0;
}
```

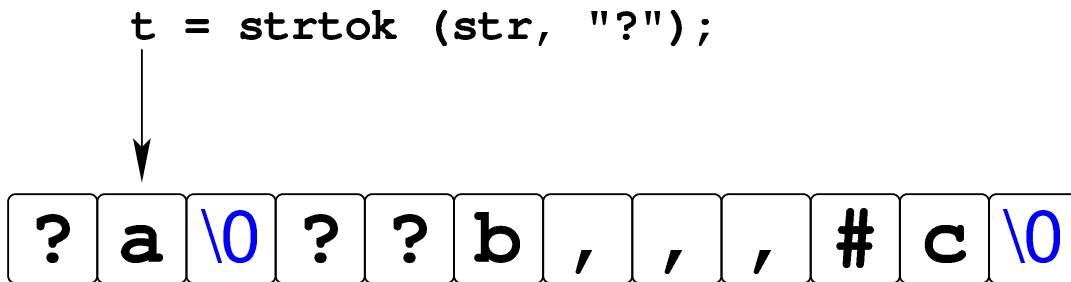
Avviando il programma si ottiene quanto già descritto dai commenti inseriti nel codice:

```
strtok: "a"
strtok: "??b"
strtok: "c"
strtok: "(null)"
```

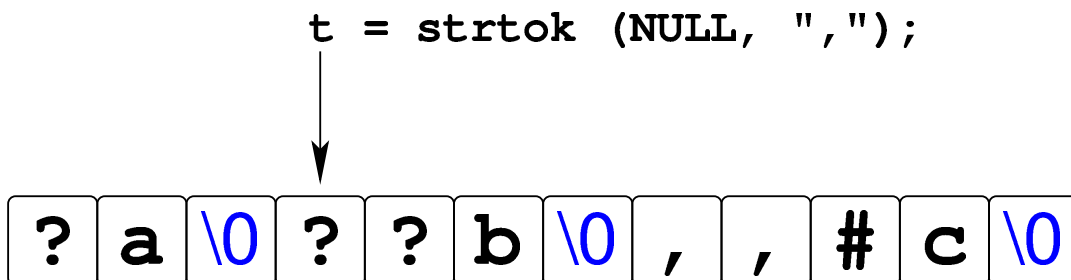
Ciò che avviene nell'esempio può essere schematizzato come segue. Inizialmente la stringa 'str' ha in memoria l'aspetto seguente:



Dopo la prima chiamata della funzione *strtok()* la stringa risulta alterata e il puntatore ottenuto raggiunge la lettera 'a':

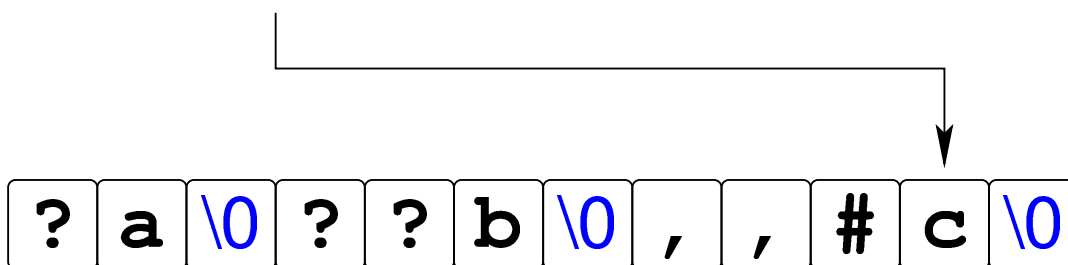


Dopo la seconda chiamata della funzione, in cui si usa il puntatore nullo per richiedere una scansione ulteriore della stringa originale, si ottiene un nuovo puntatore che, questa volta, inizia a partire dal quarto carattere, rispetto alla stringa originale, dal momento che il terzo è già stato sovrascritto da un carattere nullo:



La penultima chiamata della funzione *strtok()* raggiunge la lettera 'c' che è anche alla fine della stringa originale:

```
t = strtok (NULL, "#,");
```



L'ultimo tentativo di chiamata della funzione non può dare alcun esito, perché la stringa originale si è già conclusa.

VALORE RESTITUITO

La funzione restituisce il puntatore al prossimo «pezzo», oppure **'NULL'** se non ce ne sono più.

FILE SORGENTI

'lib/string.h' [95.20]

'lib/string/strtok.c' [95.20.23]

VEDERE ANCHE

memchr(3) [88.78], *strchr(3)* [88.114], *strpbrk(3)* [88.125], *strspn(3)* [88.127].

88.130 os32: *strtol(3)*

«

NOME

'**strtol**', '**strtoul**' - conversione di una stringa in un numero

SINTASSI

```
#include <stdlib.h>

long int strtol (const char *restrict string,
                char **restrict endptr,
                int base);

unsigned long int strtoul (const char *restrict string,
                          char **restrict endptr,
                          int base);
```

DESCRIZIONE

Le funzioni *strtol()* e *strtoul()*, convertono la stringa *string* in un numero, intendendo la sequenza di caratteri nella base di numerazione indicata come ultimo argomento (*base*). Tuttavia, la base di numerazione potrebbe essere omessa (valore zero) e in tal caso la stringa deve essere interpretata ugualmente in qualche modo: se (dopo un segno eventuale) inizia con zero seguito da un'altra cifra numerica, deve trattarsi di una sequenza ottale; se inizia con zero, quindi appare una lettera «x» deve trattarsi di un numero esadecimale; se inizia con una cifra numerica diversa da zero, deve trattarsi di un numero in base dieci.

La traduzione della stringa ha luogo progressivamente, arrestandosi quando si incontra un carattere incompatibile con la base di numerazione selezionata o stabilita automaticamente. Il valore convertito viene restituito; inoltre, se il puntatore *endptr* è valido (diverso da 'NULL'), si assegna a **endptr* la posizione raggiunta nella stringa, corrispondente al primo carattere che non può essere convertito. Pertanto, nello stesso modo, se la stringa non può

essere convertita affatto e si può assegnare qualcosa a **endptr*, alla fine, **endptr* corrisponde esattamente a *string*.

VALORE RESTITUITO

Le funzioni restituiscono il valore tratto dall'interpretazione della stringa, ammesso che sia rappresentabile, altrimenti si ottiene 'LONG_MIN' o 'LONG_MAX', a seconda dei casi, sapendo che occorre consultare la variabile *errno* per maggiori dettagli.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Argomento non valido.
ERANGE	Il valore risultante è al di fuori dell'intervallo ammissibile per la rappresentazione.

DIFETTI

La realizzazione di *strtoul()* è incompleta, in quanto si limita a utilizzare *strtol()*, convertendo il risultato in un valore senza segno.

FILE SORGENTI

'lib/stdlib.h' [95.19]

'lib/stdlib/strtol.c' [95.19.19]

'lib/stdlib/strtoul.c' [95.19.20]

88.131 os32: strtoul(3)

Vedere *strtol(3)* [88.130].

88.132 os32: strxfrm(3)

<<

NOME

‘**strxfrm**’ - *string transform*, ovvero trasformazione di una stringa

SINTASSI

```
#include <string.h>
size_t strxfrm (char *restrict dst,
                const char *restrict org,
                size_t n);
```

DESCRIZIONE

Lo scopo della funzione *strxfrm()* sarebbe quello di copiare la stringa *org*, sovrascrivendo *dst*, fino a un massimo di *n* caratteri nella destinazione, ma applicando una trasformazione relativa alla configurazione locale.

os32 non gestisce la configurazione locale, pertanto questa funzione si comporta in modo simile a *strncpy()*, con una differenza in ciò che viene restituito.

VALORE RESTITUITO

La funzione restituisce la quantità di byte utilizzati per contenere la trasformazione in *dst*, senza però contare il carattere nullo di terminazione.

FILE SORGENTI

‘lib/string.h’ [95.20]

‘lib/string/strxfrm.c’ [95.20.24]

VEDERE ANCHE

memcmp(3) [88.79], *strcmp(3)* [88.115], *strcoll(3)* [88.115].

88.133 os32: ttyname(3)



NOME

‘**ttyname**’ - determinazione del percorso del file di dispositivo di un terminale aperto

SINTASSI

```
#include <unistd.h>
char *ttyname (int fdn);
```

DESCRIZIONE

La funzione *ttyname()* richiede come unico argomento il numero che identifica il descrittore di un file. Ammesso che tale descrittore si riferisca a un terminale, la funzione restituisce il puntatore a una stringa che rappresenta il percorso del file di dispositivo corrispondente.

La stringa in questione viene modificata se si usa la funzione in altre occasioni.

VALORE RESTITUITO

La funzione restituisce il puntatore a una stringa che descrive il percorso del file di dispositivo, presunto, del terminale aperto con il numero *fdn*. Se non si tratta di un terminale, si ottiene un errore. In ogni caso, se la funzione non può restituire un’informazione corretta, produce semplicemente il puntatore nullo e aggiorna la variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EBADF	Il descrittore di file indicato non è valido.
ENOTTY	Il descrittore di file indicato non riguarda un terminale.

FILE SORGENTI

‘lib/unistd.h’ [[95.30](#)]

‘lib/unistd/ttyname.c’ [[95.30.41](#)]

VEDERE ANCHE

stat(2) [[87.55](#)], *isatty(3)* [[88.69](#)].

88.134 os32: *unsetenv(3)*

«

Vedere *setenv(3)* [[88.104](#)].

88.135 os32: *vfprintf(3)*

«

Vedere *vprintf(3)* [[88.137](#)].

88.136 os32: *vfscanf(3)*

«

Vedere *vfscanf(3)* [[88.138](#)].

88.137 os32: *vprintf(3)*

«

NOME

‘**vprintf**’, ‘**vfprintf**’, ‘**vsprintf**’, ‘**vsnprintf**’ - composizione dei dati per la visualizzazione

SINTASSI

```
#include <stdarg.h>
#include <stdio.h>
int vprintf (char *restrict format, va_list arg);
int vfprintf (FILE *fp, char *restrict format,
             va_list arg);
int vsnprintf (char *restrict string, size_t size,
              const char *restrict format, va_list ap);
int vsprintf (char *string, char *restrict format,
             va_list arg);
```

DESCRIZIONE

Le funzioni del gruppo ‘**v...printf()**’ hanno in comune lo scopo di comporre dei dati in forma di stringa, generalmente per la visualizzazione, o comunque per la fruizione a livello umano.

I dati in ingresso possono essere vari e vengono comunicati attraverso un puntatore di tipo ‘**va_list**’. Per quantificare e qualificare questi dati in ingresso, la stringa a cui punta il parametro *format*, deve contenere degli *specificatori di conversione*, oltre eventualmente ad altri caratteri. Pertanto, queste funzioni, prendono la stringa a cui punta *format*, la interpretano e determinano come scandire gli argomenti a cui fa riferimento il puntatore *arg*, quindi producono un’altra stringa, composta dalla stringa precedente, sostituendo gli specificatori di conversione con i dati a cui questi si riferiscono, secondo una forma di conversione definita dagli specificatori stessi.

In generale, le funzioni ‘**v...printf()**’ servono per realizzare le altre funzioni ‘**...printf()**’, le quali invece ricevono gli argo-

menti variabili direttamente. Per esempio, la funzione *printf()* può essere realizzata utilizzando in pratica *vprintf()*:

```
#include <stdio.h>
#include <stdarg.h>
int
printf (char *restrict format, ...)
{
    va_list ap;
    va_start (ap, format);
    return (vprintf (format, ap));
}
```

Si veda *printf(3)* [88.91], per la descrizione di come va predisposta la stringa *format*. Nella realizzazione di os32, di tutte queste funzioni, quella che compie effettivamente il lavoro di interpretazione della stringa di formato e che in qualche modo viene chiamata da tutte le altre, è soltanto *vsnprintf()*.

VALORE RESTITUITO

Le funzioni restituiscono la quantità di caratteri utilizzati nella composizione della nuova stringa, escluso il carattere nullo di terminazione.

FILE SORGENTI

‘lib/stdarg.h’ [95.1.10]

‘lib/stdio.h’ [95.18]

‘lib/stdio/FILE.c’ [95.18.1]

‘lib/stdio/vfprintf.c’ [95.18.37]

‘lib/stdio/vprintf.c’ [95.18.40]

‘lib/stdio/vsprintf.c’ [95.18.43]

‘lib/stdio/vsnprintf.c’ [95.18.42]

VEDERE ANCHE

fprintf(3) [88.91], *printf(3)* [88.91], *sprintf(3)* [88.91],
snprintf(3) [88.91], *scanf(3)* [88.102].

88.138 os32: vscanf(3)



NOME

‘**vscanf**’, ‘**vfscanf**’, ‘**vsscanf**’ - interpretazione dell’input e conversione

SINTASSI

```
#include <stdarg.h>
#include <stdio.h>

int vscanf (const char *restrict format, va_list ap);
int vfscanf (FILE *restrict fp, const char *restrict format,
             va_list ap);
int vsscanf (const char *string, const char *restrict format,
             va_list ap);
```

DESCRIZIONE

Le funzioni del gruppo ‘**v...scanf()**’ hanno in comune lo scopo di interpretare dei dati, forniti in forma di stringa, convertendoli opportunamente.

I dati in ingresso sono costituiti da una sequenza di caratteri, la quale viene fornita tramite lo standard input per *vscanf()*, tramite il flusso di file *fp* per *vfscanf()*, oppure tramite la stringa *string* per *vsscanf()*. L’interpretazione dei dati in ingresso viene guidata da una stringa di formato, costituita dal parametro *format*, per le

tre funzioni. La stringa di formato contiene degli *specificatori di conversione*, con cui si determina il tipo degli argomenti variabili a cui punta inizialmente *ap*.

Queste funzioni servono per realizzare in pratica quelle corrispondenti che hanno nomi privi della lettera «v» iniziale. Per esempio, per ottenere *scanf()* si può utilizzare *vscanf()*:

```
#include <stdio.h>
#include <stdarg.h>
int
scanf (const char *restrict format, ...)
{
    va_list ap;
    va_start (ap, format);
    return vscanf (format, ap);
}
```

Il modo in cui va predisposta la stringa di formato (*format*) è descritto in *scanf(3)* [88.102]. La funzione più importante di questo gruppo, in quanto svolge effettivamente il lavoro di interpretazione e viene chiamata, più o meno indirettamente, da tutte le altre, è *vfscanf()*, la quale però non è standard.

VALORE RESTITUITO

Le funzioni restituiscono la quantità di elementi in ingresso interpretati e assegnati correttamente: una quantità inferiore al previsto indica pertanto un errore. Se le funzioni restituiscono il valore ‘**EOF**’, si tratta di un errore, dovuto eventualmente a un problema di interpretazione del formato o a un problema di accesso al flusso di file da cui deve provenire l’input.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Gli argomenti forniti alla chiamata non sono validi per qualche ragione.
EPERM	Operazione di accesso non consentita.
EACCES	Accesso non consentito.
EBADF	Il descrittore del file a cui si riferisce il flusso, non è valido.
ERANGE	Il risultato della conversione di un intero, non può essere memorizzato nel tipo di variabile a cui si riferisce lo specificatore di conversione.

FILE SORGENTI

'lib/stdio.h' [95.18]

'lib/stdio/vfscanf.c' [95.18.38]

'lib/stdio/vscanf.c' [95.18.41]

'lib/stdio/vsscanf.c' [95.18.44]

'lib/stdio/vfsscanf.c' [95.18.39]

VEDERE ANCHE

fscanf(3) [88.102], *scanf(3)* [88.102], *sscanf(3)* [88.102],
printf(3) [88.91].

88.139 os32: *vsnprintf(3)*

Vedere *vprintf(3)* [88.137].



88.140 os32: vsprintf(3)

«

Vedere *vprintf(3)* [[88.137](#)].

88.141 os32: vsscanf(3)

«

Vedere *vsscanf(3)* [[88.138](#)].

Sezione 4: file speciali

89.1 os32: console(4)

NOME

‘/dev/console’ - file di dispositivo che rappresenta la console e le console virtuali

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
‘/dev/ console’	file di dispositivo a caratteri	5	255	0644 ₈
‘/dev/ console0’	file di dispositivo a caratteri	5	0	0644 ₈
‘/dev/ console1’	file di dispositivo a caratteri	5	1	0644 ₈
‘/dev/ console2’	file di dispositivo a caratteri	5	2	0644 ₈
‘/dev/ console3’	file di dispositivo a caratteri	5	3	0644 ₈

DESCRIZIONE

Il file di dispositivo ‘/dev/console’ rappresenta la console virtuale attiva in un certo momento; i file ‘/etc/console n ’ rappresentano la console virtuale n , dove n va da zero a quattro. I permessi di accesso a questi file di dispositivo sono limitati in modo da consentire solo al proprietario di accedere in scrittura. Tuttavia, per i file di dispositivo usati effettivamente come terminali di controllo, i permessi e la proprietà sono gestiti automaticamente dai programmi ‘**getty**’ e ‘**login**’.

VEDERE ANCHE

MAKEDEV(8) [92.6], *tty(4)* [89.14].

89.2 os32: ata(4)

«

NOME

‘/dev/ata n ’ - file di dispositivo per le unità di memorizzazione a disco ATA.

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
‘/dev/ata0’	file di dispositivo a blocchi	6	0	0644 ₈
‘/dev/ata1’	file di dispositivo a blocchi	6	1	0644 ₈
‘/dev/ata2’	file di dispositivo a blocchi	6	2	0644 ₈
‘/dev/ata3’	file di dispositivo a blocchi	6	3	0644 ₈
‘/dev/ata4’	file di dispositivo a blocchi	6	4	0644 ₈
‘/dev/ata5’	file di dispositivo a blocchi	6	5	0644 ₈
‘/dev/ata6’	file di dispositivo a blocchi	6	6	0644 ₈
‘/dev/ata7’	file di dispositivo a blocchi	7	6	0644 ₈

DESCRIZIONE

I file di dispositivo ‘/dev/ata n ’ rappresentano, ognuno, un’unità di memorizzazione a disco ATA. La prima unità è ‘/dev/ata0’, quelle successive procedono con la numerazione.

os32 gestisce solo unità a disco ATA; inoltre, non è ammissibile la suddivisione in partizioni.

VEDERE ANCHE

MAKEDEV(8) [92.6].

89.3 os32: kmem_arp(4)

NOME

‘/dev/kmem_arp’ - accesso alla memoria del kernel contenente la tabella ARP

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
‘/dev/ kmem_arp’	file di dispositivo a caratteri	4	6	0444 ₈

DESCRIZIONE

Il file di dispositivo ‘/dev/kmem_arp’ consente di accedere in lettura all’area di memoria che, nel kernel, rappresenta la tabella ARP. La tabella ARP è un array di ‘**ARP_MAX_ITEMS**’ elementi, di tipo ‘**arp_t**’, secondo le definizioni contenute nel file ‘kernel/net/arp.h’.

VEDERE ANCHE

MAKEDEV(8) [92.6], *kmem_ps(4)* [89.8], *kmem_mmp(4)* [89.6], *kmem_sb(4)* [89.10], *kmem_inode(4)* [89.5], *kmem_file(4)* [89.4].

89.4 os32: kmem_file(4)

«

NOME

‘/dev/kmem_file’ - accesso alla memoria del kernel contenente la tabella dei file

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
‘/dev/ kmem_file’	file di dispositivo a caratteri	4	5	0444 ₈

DESCRIZIONE

Il file di dispositivo ‘/dev/kmem_file’ consente di accedere in lettura all’area di memoria che, nel kernel, rappresenta la tabella dei file. La tabella dei file è un array di ‘**FILE_MAX_SLOTS**’ elementi, di tipo ‘**file_t**’, secondo le definizioni contenute nel file ‘kernel/fs.h’.

VEDERE ANCHE

MAKEDEV(8) [92.6], *kmem_ps(4)* [89.8], *kmem_mmp(4)* [89.6], *kmem_sb(4)* [89.10], *kmem_inode(4)* [89.5].

89.5 os32: kmem_inode(4)

**NOME**

‘/dev/kmem_inode’ - accesso alla memoria del kernel contenente la tabella degli inode

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
‘/dev/kmem_inode’	file di di- spositivo a caratteri	4	4	0444 ₈

DESCRIZIONE

Il file di dispositivo ‘/dev/kmem_inode’ consente di accedere in lettura all’area di memoria che, nel kernel, rappresenta la tabella degli inode. La tabella degli inode è un array di ‘**INODE_MAX_SLOTS**’ elementi, di tipo ‘**inode_t**’, secondo le definizioni contenute nel file ‘kernel/fs.h’.

VEDERE ANCHE

MAKEDEV(8) [92.6], *kmem_ps(4)* [89.8], *kmem_mmp(4)* [89.6], *kmem_sb(4)* [89.10], *kmem_file(4)* [89.4].

89.6 os32: kmem_mmp(4)

**NOME**

‘/dev/kmem_mmp’ - accesso alla memoria del kernel contenente la mappa di utilizzo della memoria

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
<code>‘/dev/ kmem_mmp’</code>	file di dispositivo a caratteri	4	2	0444 ₈

DESCRIZIONE

Il file di dispositivo `‘/dev/kmem_mmp’` consente di accedere in lettura all’area di memoria che, nel kernel, rappresenta la mappa di utilizzo della memoria.

VEDERE ANCHE

MAKEDEV(8) [92.6], *kmem_ps(4)* [89.8], *kmem_sb(4)* [89.10], *kmem_inode(4)* [89.5], *kmem_file(4)* [89.4].

89.7 os32: kmem_net(4)

«

NOME

`‘/dev/kmem_net’` - accesso alla memoria del kernel contenente la tabella delle interfacce

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
<code>‘/dev/ kmem_net’</code>	file di dispositivo a caratteri	4	7	0444 ₈

DESCRIZIONE

Il file di dispositivo ‘/dev/kmem_net’ consente di accedere in lettura all’area di memoria che, nel kernel, rappresenta la tabella delle interfacce di rete. La tabella delle interfacce è un array di ‘**NET_MAX_DEVICES**’ elementi, di tipo ‘**net_t**’, secondo le definizioni contenute nel file ‘kernel/net.h’.

VEDERE ANCHE

MAKEDEV(8) [92.6], *kmem_ps(4)* [89.8], *kmem_mmp(4)* [89.6], *kmem_sb(4)* [89.10], *kmem_inode(4)* [89.5], *kmem_file(4)* [89.4], *kmem_arp(4)* [89.3].

89.8 os32: kmem_ps(4)



NOME

‘/dev/kmem_ps’ - accesso alla memoria del kernel contenente lo stato dei processi

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
‘/dev/ kmem_ps’	file di dispositivo a caratteri	4	1	0444 ₈

DESCRIZIONE

Il file di dispositivo ‘/dev/kmem_ps’ consente di accedere in lettura all’area di memoria che, nel kernel, rappresenta la tabella dei processi. La tabella dei processi è un array di ‘**PROCESS_MAX**’ elementi, di tipo ‘**proc_t**’, secondo le definizioni contenute

nel file `'kernel/proc.h'`. Questo meccanismo viene usato dal programma `'ps'` per leggere e visualizzare lo stato dei processi.

VEDERE ANCHE

MAKEDEV(8) [92.6], *kmem_mmp(4)* [89.6], *kmem_sb(4)* [89.10], *kmem_inode(4)* [89.5], *kmem_file(4)* [89.4].

89.9 os32: kmem_route(4)

«

NOME

`'/dev/kmem_route'` - accesso alla memoria del kernel contenente la tabella degli instradamenti

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
<code>'/dev/kmem_route'</code>	file di di- spositivo a caratteri	4	8	0444 ₈

DESCRIZIONE

Il file di dispositivo `'/dev/kmem_route'` consente di accedere in lettura all'area di memoria che, nel kernel, rappresenta la tabella degli instradamenti. La tabella degli instradamenti è un array di `'ROUTE_MAX_ROUTES'` elementi, di tipo `'route_t'`, secondo le definizioni contenute nel file `'kernel/net/route.h'`.

VEDERE ANCHE

MAKEDEV(8) [92.6], *kmem_ps(4)* [89.8], *kmem_mmp(4)* [89.6], *kmem_sb(4)* [89.10], *kmem_inode(4)* [89.5], *kmem_file(4)* [89.4], *kmem_arp(4)* [89.3], *kmem_net(4)* [89.7].

89.10 os32: kmem_sb(4)

**NOME**

‘/dev/kmem_sb’ - accesso alla memoria del kernel contenente la tabella dei super blocchi

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
‘/dev/ kmem_sb’	file di dispositivo a caratteri	4	3	0444 ₈

DESCRIZIONE

Il file di dispositivo ‘/dev/kmem_sb’ consente di accedere in lettura all’area di memoria che, nel kernel, rappresenta la tabella dei super blocchi. La tabella dei super blocchi è un array di ‘**SB_MAX_SLOTS**’ elementi, di tipo ‘**sb_t**’, secondo le definizioni contenute nel file ‘kernel/fs.h’.

VEDERE ANCHE

MAKEDEV(8) [92.6], *kmem_ps(4)* [89.8], *kmem_mmp(4)* [89.6], *kmem_inode(4)* [89.5], *kmem_file(4)* [89.4].

89.11 os32: mem(4)

**NOME**

‘/dev/mem’ - file di dispositivo per l’accesso alla memoria del processo

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
'/dev/mem'	file di dispositivo a caratteri	1	1	0444 ₈

DESCRIZIONE

Il file di dispositivo '/dev/mem' consente di leggere la memoria del processo.

VEDERE ANCHE

MAKEDEV(8) [[92.6](#)].

89.12 os32: null(4)

«

NOME

'/dev/null' - file di dispositivo per la distruzione dei dati

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
'/dev/null'	file di dispositivo a caratteri	1	2	0666 ₈

DESCRIZIONE

Il file di dispositivo '/dev/null' appare in lettura come un file completamente vuoto, mentre in scrittura è un file in cui si può scrivere indefinitivamente, perdendo però i dati che vi si immettono.

VEDERE ANCHE

MAKEDEV(8) [92.6], *zero(4)* [89.15].

89.13 os32: port(4)

**NOME**

`‘/dev/port’` - file di dispositivo per accedere alle porte di I/O

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
<code>‘/dev/port’</code>	file di dispositivo a caratteri	1	3	0644 ₈

DESCRIZIONE

Il file di dispositivo `‘/dev/port’` consente di accedere alle porte di I/O. Tali porte consentono di leggere uno o al massimo due byte, pertanto la dimensione della lettura può essere `‘(size_t) 1’` oppure `‘(size_t) 2’`. Per selezionare l’indirizzo della porta occorre posizionare il riferimento interno al file a un indirizzo pari a quello della porta, prima di eseguire la lettura o la scrittura.

VEDERE ANCHE

MAKEDEV(8) [92.6], *mem(4)* [89.11].

89.14 os32: tty(4)

«

NOME

‘/dev/tty’ - file di dispositivo che rappresenta il terminale di controllo del processo

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
‘/dev/tty’	file di dispositivo a caratteri	2	0	0666 ₈

DESCRIZIONE

Il file di dispositivo ‘/dev/tty’ rappresenta il terminale di controllo del processo; in altri termini, il processo che accede al file ‘/dev/tty’, raggiunge il proprio terminale di controllo.

VEDERE ANCHE

MAKEDEV(8) [92.6], *console(4)* [89.1].

89.15 os32: zero(4)

«

NOME

‘/dev/zero’ - file di dispositivo per la produzione del valore zero

CONFIGURAZIONE

File	Tipo	Nu- mero prima- rio	Nu- mero secon- dario	Per- messi
<code>‘/dev/zero’</code>	file di dispositivo a caratteri	1	4	0666 ₈

DESCRIZIONE

Il file di dispositivo `‘/dev/zero’` appare in lettura come un file di lunghezza indefinita, contenente esclusivamente il valore zero (lo zero binario), mentre in scrittura è un file in cui si può scrivere indefinitivamente, perdendo però i dati che vi si immettono.

VEDERE ANCHE

MAKEDEV(8) [92.6], *null(4)* [89.12].

Sezione 5: formato dei file e convenzioni

90.1 os32: group(5)

NOME

‘/etc/group’ - elenco dei gruppi

DESCRIZIONE

Il file ‘/etc/group’ contiene l’elenco dei gruppi di utenti del sistema, uno per ogni riga. Le righe sono divise in quattro campi, delimitati con il carattere due punti (:), come nell’esempio seguente, che rappresenta l’impostazione predefinita di os32:

```
root:x:0:  
user:y:233:
```

I campi hanno il significato descritto nell’elenco seguente:

1. nominativo utente;
2. parola d’ordine, ma non usato, anche se è **comunque necessario scriverci qualcosa**;
3. numero GID, ovvero il numero del gruppo;
4. elenco di utenti aggregati, separati da una virgola, ma questa informazione non viene usata da os32.

Il file deve essere accessibile in lettura a tutti gli utenti.

VEDERE ANCHE

login(1) [[86.14](#)].

90.2 os32: inittab(5)

<<

NOME

`/etc/inittab` - configurazione di `init`

DESCRIZIONE

Il file `/etc/inittab` contiene la configurazione di `init`, per la definizione dei processi da avviare per la messa in funzione del sistema operativo. Il file può contenere dei commenti, preceduti dal carattere `#` e «voci» costituite da righe suddivise in quattro campi, separati da due punti (:), come nell'esempio seguente:

```
c0:1:respawn:/bin/getty /dev/console0
c1:1:respawn:/bin/getty /dev/console1
c2:1:respawn:/bin/getty /dev/console2
c3:1:respawn:/bin/getty /dev/console3
```

I campi hanno il significato descritto nell'elenco seguente:

1. codice che identifica univocamente la voce;
2. i livelli di esecuzione per cui la voce è valida;
3. l'azione da compiere sulla voce;
4. il programma da avviare, con tutte le opzioni e gli argomenti necessari.

Il programma `init` di os32 non distingue i livelli di esecuzione e considera soltanto l'azione `respawn`, con la quale si intende che `init` debba riavviare il processo, quando questo muore, o comunque quando muore quel processo che ha preso il suo posto.

VEDERE ANCHE

`init(8)` [92.4], `getty(8)` [92.2], `login(1)` [86.14].

90.3 os32: issue(5)



NOME

`‘/etc/issue’` - messaggio che precede **‘login’**

DESCRIZIONE

Il file `‘/etc/issue’` viene visualizzato da **‘getty’**, prima dell’avvio di **‘login’**. Il contenuto predefinito di questo file, per os32, è il seguente:

```
os32: a basic os. [Ctrl q], [Ctrl r], [Ctrl s], [Ctrl t] to change console.
```

Il programma **‘getty’** di os32 non è in grado di interpretare il contenuto del file, pertanto lo visualizza letteralmente; tuttavia, **‘getty’** mostra, indipendentemente dalla presenza e dal contenuto del file `‘/etc/issue’`, delle informazioni sul terminale per il quale è in funzione.

VEDERE ANCHE

getty(8) [[92.2](#)].

90.4 os32: passwd(5)



NOME

`‘/etc/passwd’` - elenco delle utenze

DESCRIZIONE

Il file `‘/etc/passwd’` contiene l’elenco degli utenti del sistema, uno per ogni riga. Le righe sono divise in sette campi, delimitati con il carattere due punti (:), come nell’esempio seguente, che rappresenta l’impostazione predefinita di os32:

```
root:ciao:0:0:root:/root:/bin/shell
user:ciao:1001:1001:test user:/home/user:/bin/shell
```

I campi hanno il significato descritto nell'elenco seguente:

1. nominativo utente;
2. parola d'ordine, in chiaro, per l'identificazione con il programma **'login'**;
3. numero UID, ovvero il numero dell'utente;
4. numero GID, ovvero il numero del gruppo;
5. descrizione dell'utenza;
6. shell.

Trattandosi di un sistema operativo elementare, la parola d'ordine appare in chiaro nel secondo campo, senza altri accorgimenti. Inoltre, il file deve essere accessibile in lettura a tutti gli utenti.

VEDERE ANCHE

login(1) [[86.14](#)].

Sezione 7: varie

91.1 os32: environ(7)

NOME

‘**environ**’ - ambiente del processo elaborativo

SINTASSI

```
extern char **environ;
```

DESCRIZIONE

La variabile *environ*, dichiarata nel file ‘`unistd.h`’, punta a un array di stringhe, ognuna delle quali rappresenta una variabile di ambiente, con il valore a lei assegnato. Pertanto, il contenuto di queste stringhe ha una forma del tipo ‘*nome=valore*’. Per esempio ‘**HOME=/home/user**’.

In generale, l’accesso diretto ai contenuti di questo array non è conveniente, in quanto sono disponibili delle funzioni che facilitano la gestione di questi dati in forma di variabili di ambiente.

Dal momento che le funzioni di accesso alle informazioni sulle variabili di ambiente sono definite nel file ‘`stdlib.h`’, la gestione effettiva dell’array di stringhe a cui punta *environ* è inserita nei file contenuti nella directory ‘`lib/stdlib/`’ di os32. Per la precisione, nel file ‘`lib/stdlib/environment.c`’ si dichiara l’array di caratteri `_environment_table[][]` e array di puntatori a caratteri `_environment[]`:

```
char  _environment_table[ARG_MAX/32][ARG_MAX/16];  
char *_environment[ARG_MAX/32+1];
```

L'array ***_environment_table***[][] viene inizializzato con lo stato delle variabili di ambiente ereditate con l'avvio del processo; inoltre, all'array ***_environment***[] vengono assegnati i puntatori alle varie stringhe che si possono estrapolare da ***_environment_table***[][]. Questo lavoro iniziale avviene per opera della funzione ***_environment_setup***(), la quale viene avviata a sua volta dal file 'crt0.s'. Successivamente, nello stesso file 'crt0.s', viene copiato l'indirizzo dell'***_environment***[] nella variabile ***environ***, di cui sopra.

FILE SORGENTI

'lib/unistd.h' [95.30]
'lib/stdlib.h' [95.19]
'lib/unistd/environ.c' [95.30.9]
'applic/crt0.mer.s' [96.1.12]
'applic/crt0.sep.s' [96.1.13]
'lib/stdlib/environment.c' [95.19.8]
'lib/stdlib/getenv.c' [95.19.10]
'lib/stdlib/putenv.c' [95.19.15]
'lib/stdlib/setenv.c' [95.19.18]
'lib/stdlib/unsetenv.c' [95.19.21]

VEDERE ANCHE

getenv(3) [88.52], *putenv*(3) [88.94], *setenv*(3) [88.104],
unsetenv(3) [88.104].

91.2 os32: socket(7)

**NOME**

‘**socket**’ - gestione dei socket

DESCRIZIONE

Il sistema os32 gestisce esclusivamente socket di dominio Internet, con indirizzi IPv4. Per creare e gestire una connessione si usano le funzioni seguenti:

Funzione	Scopo
<code>socket ()</code>	Crea la terminazione locale di una connessione, ma senza attribuire indirizzi o porte [87.54].
<code>connect ()</code>	Connette un socket locale a un socket remoto [87.11].
<code>bind ()</code>	Attribuisce a un socket locale un indirizzo [87.4].
<code>listen ()</code>	Mette un socket locale in ascolto, attendendo una richiesta di connessione da un socket remoto [87.31].
<code>accept ()</code>	Recepisce una richiesta di connessione ricevuta da un socket locale, generando con questa un nuovo socket locale connesso a quello da cui ha avuto origine la richiesta [87.3].
<code>send ()</code>	Invia dati attraverso una connessione [87.45].
<code>recvfrom ()</code>	Riceve dati attraverso una connessione [87.40].
<code>close ()</code>	Chiude un socket locale [87.10].

Quando si crea un socket per poi contattare un socket remoto in attesa di connessioni, è possibile utilizzare le funzioni *socket()* e *connect()*, senza avvalersi di *bind()*, lasciando che le informazioni mancanti nel socket locale siano determinate o fissate automaticamente.

Quando si crea un socket da mettere in ascolto, dopo la funzione *socket()* è necessario utilizzare *bind()* per determinare i parametri locali; poi, se si tratta del protocollo TCP si usa la funzione *listen()* per attendere una richiesta di connessione da un socket remoto. Quindi, per recepire una richiesta di connessione si usa la funzione *accept()*, con cui si genera un nuovo socket locale connesso a quello remoto che ha emesso la richiesta.

Quando il socket deve essere in ascolto, ma in attesa di pacchetti UDP, dopo l'utilizzo della funzione *bind()* si può passare subito all'uso di *recvfrom()*.

VEDERE ANCHE

socket(2) [87.54], *accept(2)* [87.3], *bind(2)* [87.4], *connect(2)* [87.11], *listen(2)* [87.31].

91.3 os32: undocumented(7)

«

Questa sezione ha il solo scopo di raccogliere i riferimenti ipertestuali dei listati che, per qualche ragione, sono privi di una documentazione specifica.

Sezione 8: comandi per l'amministrazione del sistema

92.1 os32: arp(8)

NOME

'arp' - mostra la tabella ARP

SINTASSI

```
arp
```

DESCRIZIONE

Il programma **'arp'** consente di visualizzare la tabella ARP (*Address resolution protocol*, ma senza la possibilità di potervi intervenire per modificarla. Pertanto, l'uso del programma è ammissibile per qualunque utente.

os32 fornisce l'accesso alla tabella ARP attraverso il file di dispositivo **'/dev/kmem_arp'**, da cui attinge il programma **'arp'**.

FILE

'/dev/kmem_arp'

È il file di dispositivo attraverso il quale è possibile leggere la tabella ARP del kernel di os32.

FILE SORGENTI

'applic/crt0.mer.s' [96.1.12]

'applic/crt0.sep.s' [96.1.13]

'applic/arp.c' [96.1.4]

VEDERE ANCHE

kmem_arp(4) [89.3], *ipconfig(8)* [92.5], *route(8)* [92.9].

92.2 os32: getty(8)

<<

NOME

‘**getty**’ - predisposizione di un terminale e avvio di ‘**login**’

SINTASSI

```
getty terminale
```

DESCRIZIONE

Il programma ‘**getty**’ viene avviato da ‘**init**’ per predisporre il terminale, ripristinando anche i permessi predefiniti, e per avviare successivamente il programma ‘**login**’. Prima di avviare ‘**login**’, ‘**getty**’ visualizza il contenuto del file ‘`/etc/issue`’, se disponibile, inoltre mostra almeno l’indicazione del terminale attuale. Va osservato che questa realizzazione di ‘**getty**’ lascia a ‘**login**’ il compito di chiedere l’inserimento del nominativo utente.

FILE

‘`/etc/issue`’

‘**getty**’ visualizza il contenuto di questo file prima di avviare ‘**login**’.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [96.1.12]

‘`applic/crt0.sep.s`’ [96.1.13]

'`applic/getty.c`' [96.1.16]

VEDERE ANCHE

`login(1)` [86.14], `issue(5)` [90.3].

92.3 os32: http(8)



NOME

'`http`' - servente HTTP elementare

SINTASSI

```
http porta directory_radice
```

DESCRIZIONE

Il programma '`http`' si comporta come servente HTTP, in ascolto nella porta indicata come primo argomento, per pubblicare i file che si trovano a partire dalla `directory` specificata come secondo argomento della riga di comando.

Dal momento che `os32` non dispone di una shell in grado di interpretare script, il modo migliore per avviare il servizio è quello di inserire la richiesta di avvio del programma '`http`' nel file '`/etc/inittab`', con una direttiva simile a quella seguente:

```
s0:1:respawn:/bin/http 80 /var/www
```

FILE SORGENTI

'`applic/crt0.mer.s`' [96.1.12]

'`applic/crt0.sep.s`' [96.1.13]

'`applic/http.c`' [96.1.17]

VEDERE ANCHE

inittab(5) [90.2], *arp(8)* [92.1], *ipconfig(8)* [92.9], *route(8)* [92.9], *ping(8)* [92.8], *nc(8)* [86.20].

92.4 os32: init(8)

<<

NOME

‘**init**’ - progenitore di tutti gli altri processi

SINTASSI

```
init
```

DESCRIZIONE

Il programma ‘**init**’ viene avviato dal kernel (deve trattarsi precisamente del file ‘`/bin/init`’) come primo e unico processo figlio del kernel stesso. Pertanto, ‘**init**’ deve assumere il numero PID uno.

Questa realizzazione di ‘**init**’ si limita a leggere il file ‘`/etc/inittab`’ per determinare quali programmi figli avviare, senza poter distinguere da diversi livelli di esecuzione. In pratica, all’interno di questo file si indica l’uso di ‘**getty**’, per la gestione dei terminali disponibili.

FILE

‘`/etc/inittab`’

Contiene l’indicazione dei processi che ‘**init**’ deve avviare.

DIFETTI

Con os32 non è possibile associare ai segnali un’azione diversa da quella predefinita; quindi ‘**init**’ non può essere informato

dell'intenzione di arrestare il sistema. Pertanto, tale funzionalità non è stata realizzata nella versione di **'init'** di os32.

FILE SORGENTI

'`applic/crt0.mer.s`' [96.1.12]

'`applic/crt0.sep.s`' [96.1.13]

'`applic/init.c`' [96.1.18]

VEDERE ANCHE

inittab(5) [90.2].

92.5 os32: ipconfig(8)



NOME

'ipconfig' - mostra la configurazione delle interfacce di rete

SINTASSI

```
ipconfig
```

DESCRIZIONE

Il programma **'ipconfig'** consente di visualizzare la tabella delle interfacce di rete, con la loro configurazione, ma senza la possibilità di potervi intervenire per modificarla (la configurazione delle interfacce di rete avviene esclusivamente attraverso le opzioni di avvio del kernel). Pertanto, l'uso del programma è ammissibile per qualunque utente.

Va osservato che l'interfaccia **'net0'** è costituita sempre dal dispositivo interno associato all'indirizzo 127.0.0.1 (*loopback*).

os32 fornisce l'accesso alla tabella delle interfacce di rete attraverso il file di dispositivo `/dev/kmem_net`, da cui attinge il programma `ipconfig`.

FILE

`/dev/kmem_net`

È il file di dispositivo attraverso il quale è possibile leggere la tabella delle interfacce del kernel di os32.

FILE SORGENTI

`applic/crt0.mer.s` [96.1.12]

`applic/crt0.sep.s` [96.1.13]

`applic/ipconfig.c` [96.1.19]

VEDERE ANCHE

`kmem_net(4)` [89.7], `arp(8)` [92.1], `route(8)` [92.9].

92.6 os32: MAKEDEV(8)

«

NOME

MAKEDEV - creazione dei file di dispositivo

SINTASSI

```
MAKEDEV
```

DESCRIZIONE

MAKEDEV è un programma che crea, nella directory corrente, tutti i file di dispositivo previsti per os32. Tali file devono trovarsi normalmente nella directory `/dev/`, pertanto, prima di usa-

re **'MAKEDEV'** è necessario che la directory corrente corrisponda precisamente a tale posizione.

OPZIONI

Non sono previste opzioni per l'uso di **'MAKEDEV'**, dal momento che vengono creati tutti i file di dispositivo, considerato il loro numero limitato.

NOTE

Tradizionalmente **'MAKEDEV'** viene realizzato in forma di script, ma os32 non dispone di una shell adeguata e non è possibile utilizzare script.

FILE SORGENTI

'`applic/crt0.mer.s`' [96.1.12]

'`applic/crt0.sep.s`' [96.1.13]

'`lib/sys/os32.h`' [95.21]

'`applic/MAKEDEV.c`' [96.1.1]

92.7 os32: mount(8)



NOME

'mount', **'umount'** - innesto e distacco di un file system

SINTASSI

```
mount dispositivo dir_innesto [opzioni]
```

```
umount directory
```

DESCRIZIONE

‘**mount**’ innesta il file system contenuto nell’unità di memorizzazione rappresentata dal file di dispositivo che va indicato come primo argomento, nella directory che appare come secondo argomento. Eventualmente si possono specificare delle opzioni di innesto, come terzo argomento.

‘**umount**’ stacca il file system innestato precedentemente nella directory indicata come unico argomento del comando.

OPZIONI DI INNESTO

Opzione	Descrizione
ro	Innesta il file system in sola lettura.
rw	Innesta il file system in lettura e scrittura. Si tratta comunque del comportamento predefinito, in mancanza di un’opzione contraria.

DIFETTI

Non viene preso in considerazione un eventuale file ‘/etc/fstab’; inoltre, l’utente non può conoscere lo stato degli innesti già in essere e, a questo proposito, l’uso di ‘**mount**’ senza argomenti produce semplicemente un errore.

FILE SORGENTI

‘`applic/crt0.mer.s`’ [[96.1.12](#)]

‘`applic/crt0.sep.s`’ [[96.1.13](#)]

‘`applic/mount.c`’ [[96.1.28](#)]

‘`applic/umount.c`’ [[96.1.53](#)]

92.8 os32: ping(8)



NOME

'ping' - invio di richieste di eco: ICMP ECHO_REQUEST.

SINTASSI

```
ping indirizzo_ipv4
```

DESCRIZIONE

Il programma **'ping'** invia una richiesta di eco (ICMP ECHO_REQUEST) alla destinazione indicata attraverso un indirizzo IPv4. Se si ottiene risposta, il programma termina con successo, altrimenti si ottiene un errore; in ogni caso, viene eseguito un solo tentativo.

Per utilizzare **'ping'** è necessario operare in qualità di utente **'root'**.

FILE SORGENTI

`'applic/crt0.mer.s'` [[96.1.12](#)]

`'applic/crt0.sep.s'` [[96.1.13](#)]

`'applic/ping.c'` [[96.1.30](#)]

VEDERE ANCHE

arp(8) [[92.1](#)], *ipconfig(8)* [[92.9](#)], *route(8)* [[92.9](#)], *http(8)* [[92.3](#)], *nc(8)* [[86.20](#)].

92.9 os32: route(8)



NOME

‘**route**’ - mostra la tabella degli instradamenti

SINTASSI

```
route
```

DESCRIZIONE

Il programma ‘**route**’ consente di visualizzare la tabella degli instradamenti, ma senza la possibilità di potervi intervenire per modificarla (la configurazione delle interfacce di rete e degli instradamenti avviene esclusivamente attraverso le opzioni di avvio del kernel). Pertanto, l’uso del programma è ammissibile per qualunque utente.

os32 fornisce l’accesso alla tabella degli instradamenti attraverso il file di dispositivo ‘/dev/kmem_route’, da cui attinge il programma ‘**route**’.

FILE

‘/dev/kmem_route’

È il file di dispositivo attraverso il quale è possibile leggere la tabella degli instradamenti del kernel di os32.

FILE SORGENTI

‘applic/crt0.mer.s’ [96.1.12]

‘applic/crt0.sep.s’ [96.1.13]

‘applic/route.c’ [96.1.34]

VEDERE ANCHE

kmem_route(4) [89.9], *arp(8)* [92.1], *ipconfig(8)* [92.5].

92.10 os32: umount(8)

Vedere *mount(8)* [92.7].



Sezione 9: kernel



93.1	os32: arp(9)	795
93.2	os32: ata(9)	797
93.3	os32: blk(9)	802
93.3.1	os32: blk_ata(9)	802
93.3.2	os32: blk_cache_check(9)	803
93.3.3	os32: blk_cache_init(9)	805
93.3.4	os32: blk_cache_read(9)	806
93.3.5	os32: blk_cache_save(9)	807
93.4	os32: dev(9)	808
93.4.1	os32: dev_io(9)	815
93.4.2	os32: dev_dm(9)	816
93.4.3	os32: dev_ata(9)	817
93.4.4	os32: dev_kmem(9)	818
93.4.5	os32: dev_mem(9)	819
93.4.6	os32: dev_tty(9)	822
93.5	os32: dm(9)	823
93.6	os32: fs(9)	823
93.6.1	os32: fd_dup(9)	830
93.6.2	os32: fd_reference(9)	832
93.6.3	os32: fs_init(9)	834
93.6.4	os32: file_pipe_make(9)	834
93.6.5	os32: file_reference(9)	836

93.6.6	os32: file_stdio_dev_make(9)	837
93.6.7	os32: inode_alloc(9)	839
93.6.8	os32: inode_check(9)	840
93.6.9	os32: inode_dir_empty(9)	843
93.6.10	os32: inode_file_read(9)	844
93.6.11	os32: inode_file_write(9)	846
93.6.12	os32: inode_free(9)	848
93.6.13	os32: inode_fzones_read(9)	849
93.6.14	os32: inode_fzones_write(9)	851
93.6.15	os32: inode_get(9)	851
93.6.16	os32: inode_pipe_make(9)	853
93.6.17	os32: inode_pipe_read(9)	854
93.6.18	os32: inode_pipe_write(9)	856
93.6.19	os32: inode_print(9)	858
93.6.20	os32: inode_put(9)	858
93.6.21	os32: inode_reference(9)	860
93.6.22	os32: inode_save(9)	862
93.6.23	os32: inode_stdio_dev_make(9)	863
93.6.24	os32: inode_truncate(9)	864
93.6.25	os32: inode_zone(9)	865
93.6.26	os32: sb_inode_status(9)	867
93.6.27	os32: sb_mount(9)	869
93.6.28	os32: sb_print(9)	872
93.6.29	os32: sb_reference(9)	872
93.6.30	os32: sb_save(9)	874

93.6.31	os32: sb_zone_status(9)	875
93.6.32	os32: sock_free_port(9)	875
93.6.33	os32: sock_reference(9)	876
93.6.34	os32: zone_alloc(9)	878
93.6.35	os32: zone_free(9)	879
93.6.36	os32: zone_print(9)	880
93.6.37	os32: zone_read(9)	880
93.6.38	os32: path_device(9)	882
93.6.39	os32: path_fix(9)	884
93.6.40	os32: path_full(9)	885
93.6.41	os32: path_inode(9)	886
93.6.42	os32: path_inode_link(9)	888
93.7	os32: ibm_i386(9)	891
93.8	os32: icmp(9)	897
93.9	os32: ip(9)	897
93.10	os32: kbd(9)	899
93.11	os32: lib_k(9)	900
93.12	os32: lib_s(9)	901
93.13	os32: main(9)	902
93.14	os32: memory(9)	902
93.15	os32: multiboot(9)	905
93.16	os32: ne2k(9)	907

93.17	os32: net(9)	909
93.18	os32: part(9)	912
93.19	os32: pci(9)	912
93.20	os32: proc(9)	913
93.20.1	os32: proc_available(9)	913
93.20.2	os32: proc_dump_memory(9)	914
93.20.3	os32: proc_init(9)	915
93.20.4	os32: proc_print(9)	918
93.20.5	os32: proc_reference(9)	918
93.20.6	os32: proc_sch_net(9)	920
93.20.7	os32: proc_sch_signals(9)	921
93.20.8	os32: proc_sch_terminals(9)	922
93.20.9	os32: proc_sch_timers(9)	923
93.20.10	os32: proc_scheduler(9)	924
93.20.11	os32: proc_sig_chld(9)	927
93.20.12	os32: proc_sig_cont(9)	928
93.20.13	os32: proc_sig_core(9)	930
93.20.14	os32: proc_sig_handler(9)	932
93.20.15	os32: proc_sig_ignore(9)	933
93.20.16	os32: proc_sig_off(9)	934
93.20.17	os32: proc_sig_on(9)	934
93.20.18	os32: proc_sig_status(9)	936
93.20.19	os32: proc_sig_stop(9)	937
93.20.20	os32: proc_sig_term(9)	938

93.20.21	os32: proc_sys_exec(9)	940
93.20.22	os32: proc_timer_init(9)	944
93.20.23	os32: proc_wakeup(9)	945
93.20.24	os32: proc_wakeup_pipe_read(9)	947
93.20.25	os32: proc_wakeup_pipe_write(9)	947
93.20.26	os32: proc_wakeup_terminal(9)	947
93.20.27	os32: ptr(9)	947
93.20.28	os32: sysroutine(9)	948
93.21	os32: route(9)	950
93.22	os32: screen(9)	952
93.23	os32: tcp(9)	955
93.24	os32: tty(9)	959
arp.h	795	ata.h	797
blk.h	802	blk_ata()	802
blk_cache_check()	803	blk_cache_init()	805
blk_cache_read()	806	blk_cache_save()	806
dev.h	808	dev_ata()	817
dev_ata()	817	dev_dm()	816
dev_io()	815	dev_kmem()	818
dev_mem()	819	dev_tty()	822
dm.h	823	fd_dup()	830
fd_reference()	832	file_pipe_make()	834
file_reference()	836	file_stdio_dev_make()	837
fs.h	823	fs_init()	834
ibm_i386.h	891	icmp.h	897
inode_alloc()	839	inode_check()	840
inode_dir_empty()	843	inode_file_read()	844
inode_file_write()	846	inode_free()	848
inode_fzones_read()	849	inode_fzones_write()	849
inode_get()	851	inode_pipe_make()	853
inode_pipe_read()	854	inode_pipe_write()	856

inode_print() 858 inode_put() 858
 inode_reference() 860 inode_save() 862
 inode_stdio_dev_make() 863 inode_truncate() 864
 inode_zone() 865 ip.h 897 kbd.h 899 lib_k.h 900
 lib_s.h 901 main.h 902 memory.h 902 multiboot.h 905
 ne2k.h 907 net.h 909 part.h 912 path_device() 882
 path_fix() 884 path_full() 885 path_inode() 886
 path_inode_link() 888 pci.h 912 proc.h 913
 proc_available() 913 proc_dump_memory() 914
 proc_init() 915 proc_print() 918
 proc_reference() 918 proc_scheduler() 924
 proc_sch_net() 920 proc_sch_signals() 921
 proc_sch_terminals() 922 proc_sch_timers() 923
 proc_sig_chld() 927 proc_sig_cont() 928
 proc_sig_core() 930 proc_sig_handler() 932
 proc_sig_ignore() 933 proc_sig_off() 934
 proc_sig_on() 934 proc_sig_status() 936
 proc_sig_stop() 937 proc_sig_term() 938
 proc_sys_exec() 940 proc_timer_init()
 944 proc_wakeup_pipe_read()
 945 proc_wakeup_pipe_write() 945
 proc_wakeup_terminal() 945 ptr() 947 route.h 950
 sb_inode_status() 867 sb_mount() 869 sb_print()
 872 sb_reference() 872 sb_save() 874
 sb_zone_status() 867 screen.h 952
 sock_free_port() 875 sock_reference() 876
 sysroutine() 948 s_brk() 901 s_chdir() 901
 s_chmod() 901 s_chown() 901 s_clock() 901
 s_close() 901 s_dup() 901 s_dup2() 901 s_fchmod()

[901](#) s_fchown() [901](#) s_fcntl() [901](#) s_fork() [901](#)
[s_fstat\(\)](#) [901](#) s_kill() [901](#) s_link() [901](#)
[s_longjmp\(\)](#) [901](#) s_lseek() [901](#) s_mkdir() [901](#)
[s_mknod\(\)](#) [901](#) s_mount() [901](#) s_open() [901](#) s_pipe()
[901](#) s_read() [901](#) s_sbrk() [901](#) s_setegid() [901](#)
[s_seteuid\(\)](#) [901](#) s_setgid() [901](#) s_setjmp() [901](#)
[s_setuid\(\)](#) [901](#) s_signal() [901](#) s_stat() [901](#)
[s_stime\(\)](#) [901](#) s_tcgetattr() [901](#) s_tcsetattr() [901](#)
[s_time\(\)](#) [901](#) s_umount() [901](#) s_unlink() [901](#)
[s_wait\(\)](#) [901](#) s_write() [901](#) s__exit() [901](#) tcp.h [955](#)
tty.h [959](#) zone_alloc() [878](#) zone_free() [878](#)
zone_print() [880](#) zone_read() [880](#) zone_write() [880](#)

93.1 os32: arp(9)

Il file ‘kernel/net/arp.h’ [\[94.12.1\]](#) descrive le funzioni per la gestione della tabella ARP, per la trasformazione degli indirizzi IPv4 in indirizzi Ethernet. «

Per la descrizione sulla gestione della tabella ARP da parte di os32, si rimanda alla sezione [84.9.3](#). La tabella successiva che sintetizza l’uso delle funzioni di questo gruppo, è tratta da lì.

Tabella 84.123. Funzioni per la gestione della tabella ARP, contenute nella directory ‘kernel/net/arp/’.

Funzione	Descrizione
<pre>void arp_init (void);</pre>	<p>Azzera completamente la tabella ARP: si usa una volta sola all’avvio della gestione della rete [94.12.4].</p>
<pre>void arp_clean (void);</pre>	<p>Azzera le voci della tabella ARP che risultano troppo vecchie e che devono essere rinnovate [94.12.2].</p>
<pre>int arp_index (unsigned char <i>mac</i>[6], h_addr_t <i>ip</i>);</pre>	<p>Restituisce l’indice della tabella ARP, corrispondente all’indirizzo Ethernet o all’indirizzo IPv4 fornito [94.12.3].</p>
<pre>arp_t *arp_reference (void);</pre>	<p>Restituisce il puntatore a un elemento della tabella ARP contenente la voce più vecchia, allo scopo presumibile di riutilizzarla per un indirizzo nuovo [94.12.7].</p>
<pre>void arp_request (h_addr_t <i>ip</i>);</pre>	<p>Invia una richiesta ARP, preparando il pacchetto relativo e inviandolo attraverso la funzione <i>ethernet_tx()</i> [94.12.8].</p>

Funzione	Descrizione
<pre>int arp_rx (int <i>n</i>, int <i>f</i>);</pre>	<p>Legge dalla tabella delle interfacce il pacchetto individuato dall'indice <i>n</i> per l'interfaccia e dall'indice <i>f</i> per la trama relativa. Il pacchetto in questione deve essere relativo al protocollo ARP: se si tratta di una richiesta, provvede a inviare una risposta, se invece si tratta di una risposta, allora aggiorna la tabella ARP [94.12.9].</p>

93.2 os32: ata(9)

Il file 'kernel/driver/ata.h' [94.4.3] descrive le funzioni per la gestione delle unità a disco PATA. «

Per la descrizione dell'organizzazione della gestione delle unità PATA di os32, si rimanda alla sezione 84.7.7. La tabella successiva che sintetizza l'uso delle funzioni di questo gruppo, è tratta da lì.

Tabella 84.94. Funzioni per la gestione delle unità PATA, dichiarate nel file di intestazione 'kernel/driver/ata.h' e descritte nei file contenuti nella directory 'kernel/driver/ata/'. Le funzioni sono raggruppate in insiemi logici.

Funzione	Descrizione
<pre>void ata_init (void);</pre>	Inizializza la gestione delle unità PATA, predisponendo i contenuti della tabella <i>ata_table[]</i> , verificando la presenza delle unità. Questa funzione viene usata una volta sola, nella funzione <i>proc_init()</i> .
<pre>void ata_reset (int <i>drive</i>);</pre>	Azzera lo stato di funzionamento dell'unità PATA specificata.
<pre>int ata_valid (int <i>drive</i>);</pre>	Verifica se l'unità richiesta è presente effettivamente. In caso di successo restituisce il valore zero, altrimenti si ottiene -1.

Funzione	Descrizione
<pre>int ata_cmd_identify_device (int <i>drive</i>, void *<i>buffer</i>);</pre>	<p>Richiede all'unità specificata le informazioni sulla sua identificazione. Se l'unità è presente, in corrispondenza del puntatore fornito si ottengono le informazioni nello spazio di un settore (ATA_SECTOR_SIZE); l'analisi successiva di questi dati può dare maggiori informazioni sull'unità.</p>
<pre>int ata_cmd_read_sectors (int <i>drive</i>, unsigned int <i>sector</i>, unsigned char <i>count</i>, void *<i>buffer</i>);</pre>	<p>Legge dall'unità <i>drive</i>, a partire dal settore <i>sector</i>, una quantità pari a <i>count</i> settori, mettendo il risultato a partire dall'indirizzo di memoria <i>buffer</i>. Se <i>count</i> fosse pari a zero, si intenderebbero 256 settori. Se l'operazione fallisce, restituisce un valore negativo.</p>

Funzione	Descrizione
<pre>int ata_cmd_write_sectors (int <i>drive</i>, unsigned int <i>sector</i>, unsigned char <i>count</i>, void *<i>buffer</i>);</pre>	<p>Scrive nell'unità <i>drive</i>, a partire dal settore <i>sector</i>, una quantità pari a <i>count</i> settori, leggendoli a partire dall'indirizzo di memoria <i>buffer</i>. Se <i>count</i> fosse pari a zero, si intenderebbero 256 settori. Se l'operazione fallisce, restituisce un valore negativo.</p>
<pre>int ata_device (int <i>drive</i>, unsigned int <i>sector</i>);</pre>	<p>Imposta il registro <i>device</i> dell'unità PATA specificata, con l'indicazione di un numero di settore.</p>

Funzione	Descrizione
<pre>int ata_rdy (int <i>drive</i>, clock_t <i>timeout</i>);</pre>	<p>Attende che l'unità <i>drive</i> sia pronta, purché ciò avvenga entro il tempo <i>timeout</i>. Se l'operazione ha successo, la funzione restituisce zero, altrimenti dà un valore negativo.</p>
<pre>int ata_drq (int <i>drive</i>, clock_t <i>timeout</i>);</pre>	<p>Attende che l'unità <i>drive</i> sia pronta a ricevere dati, purché ciò avvenga entro il tempo <i>timeout</i>. Se l'operazione ha successo, la funzione restituisce zero, altrimenti dà un valore negativo.</p>

Funzione	Descrizione
<pre>int ata_lba28 (int <i>drive</i>, unsigned int <i>sector</i>, unsigned char <i>count</i>);</pre>	<p>Invia all'unità <i>drive</i> la prima parte di un comando, in cui sono contenute le coordinate LBA28.</p>

Funzione	Descrizione
<pre>int ata_read_sector (int <i>drive</i>, unsigned int <i>sector</i>, void *<i>buffer</i>);</pre>	<p>È una macroistruzione che legge dall'unità <i>drive</i>, il settore <i>sector</i>, mettendo il risultato a partire dall'indirizzo di memoria <i>buffer</i>. La macroistruzione si avvale praticamente della funzione <i>ata_cmd_read_sectors()</i>, per leggere un solo settore.</p>
<pre>int ata_write_sector (int <i>drive</i>, unsigned int <i>sector</i>, void *<i>buffer</i>);</pre>	<p>È una macroistruzione che scrive nell'unità <i>drive</i>, il settore <i>sector</i>, traendo i dati dall'indirizzo di memoria <i>buffer</i>. La macroistruzione si avvale praticamente della funzione <i>ata_cmd_write_sectors()</i>, per scrivere un solo settore.</p>

93.3 os32: blk(9)

«

Il file ‘kernel/blk.h’ [94.2] descrive ciò che serve per la gestione dei blocchi di dati, in relazione ai dispositivi di memorizzazione a blocchi.

93.3.1 os32: blk_ata(9)

«

NOME

‘**blk_ata**’ - interfaccia di accesso ai dispositivi di memorizzazione PATA

SINTASSI

```
<kernel/blk.h>
void *blk_ata (dev_t device, int rw, unsigned int n,
               void *buffer);
```

ARGOMENTI

Argomento	Descrizione
dev_t <i>device</i>	Dispositivo, in forma numerica.
int <i>rw</i>	Può assumere i valori ‘DEV_READ’ o ‘DEV_WRITE’, per richiedere rispettivamente un accesso in lettura oppure in scrittura.
unsigned int <i>n</i>	Numero del blocco da leggere o scrivere, riferito all’unità complessiva.
void * <i>buffer</i>	Origine dei dati da trasferire in caso di scrittura (per la lettura questa informazione non viene usata).

DESCRIZIONE

La funzione *blk_ata()* è un'interfaccia per l'accesso alle unità PATA, un blocco alla volta, che si avvale di una memoria *cache* per ridurre gli accessi fisici ripetuti agli stessi blocchi.

VALORE RESTITUITO

La funzione restituisce il puntatore a un'area di memoria contenente il blocco di dati letto o scritto. In caso di errore restituisce il puntatore nullo e aggiorna la variabile *errno* del kernel. Il blocco in questione si riferisce all'unità complessiva, pertanto, se originariamente si faceva riferimento a un dispositivo di una partizione, prima di arrivare a questa funzione il blocco relativo della partizione deve essere stato convertito in un blocco assoluto dell'unità complessiva.

ERRORI

Valore di <i>errno</i>	Significato
E_HARDWARE_FAULT	Errore nell'hardware PATA.
E_DRIVER_FAULT	Errore nella gestione software dell'unità PATA.
ETIME	Tempo scaduto.

FILE SORGENTI

'kernel/blk.h' [94.2]

'kernel/blk/blk_ata.c' [94.2.1]

VEDERE ANCHE

dev_dm(9) [93.4.2], *dev_ata(9)* [93.4.3], *blk_cache_read(9)* [93.3.4], *blk_cache_save(9)* [93.3.4].

93.3.2 os32: blk_cache_check(9)



NOME

‘**blk_cache_check**’ - verifica della validità del contenuto della tabella dei blocchi conservati in memoria

SINTASSI

```
<kernel/blk.h>  
void blk_cache_check (void);
```

DESCRIZIONE

La funzione *blk_cache_check()* serve a verificare che la tabella dei blocchi conservati in memoria (*blk_table[]*) contenga dati validi, per ciò che riguarda le età degli stessi. In pratica, il valore dell’età dei blocchi deve essere sequenziale, iniziando da zero e terminando con il valore massimo consentito: non ci devono essere valori mancanti e non ci devono essere valori doppi. Questa funzione serve solo a titolo diagnostico e in pratica non viene usata.

VALORE RESTITUITO

La funzione non restituisce alcunché; se viene usata e se individua errori, questi vengono visualizzati attraverso la funzione *k_printf()*.

FILE SORGENTI

‘kernel/blk.h’ [94.2]

‘kernel/blk/blk_cache_check.c’ [94.2.2]

VEDERE ANCHE

blk_cache_init(9) [93.3.3], *blk_cache_read*(9) [93.3.4],
blk_cache_save(9) [93.3.4].

93.3.3 os32: *blk_cache_init*(9)



NOME

‘**blk_cache_init**’ - inizializzazione della tabella dei blocchi conservati in memoria

SINTASSI

```
<kernel/blk.h>  
void blk_cache_init (void);
```

DESCRIZIONE

La funzione *blk_cache_init*() serve a inizializzare la tabella dei blocchi conservati in memoria (*blk_table[]*), assegnando anche il valore dell’età in modo progressivo, da zero fino al valore massimo consentito. Questa funzione viene usata una volta sola, prima che i dispositivi di memorizzazione a blocchi possano avvalersi della tabella stessa.

VALORE RESTITUITO

La funzione non restituisce alcunché.

FILE SORGENTI

‘kernel/blk.h’ [94.2]

‘kernel/blk/blk_cache_init.c’ [94.2.3]

VEDERE ANCHE

blk_cache_check(9) [93.3.2], *blk_cache_read*(9) [93.3.4],
blk_cache_save(9) [93.3.4].

93.3.4 os32: *blk_cache_read*(9)

«

NOME

‘**blk_cache_read**’, ‘**blk_cache_save**’, - lettura e scrittura nella tabella dei blocchi

SINTASSI

```
<kernel/blk.h>
void *blk_cache_read (dev_t device, unsigned int n);
void *blk_cache_save (dev_t device, unsigned int n,
                     void *block);
```

ARGOMENTI

Argomento	Descrizione
dev_t <i>device</i>	Dispositivo, in forma numerica.
unsigned int <i>n</i>	Numero del blocco da leggere o scrivere, riferito all'unità complessiva.
void * <i>block</i>	Origine dei dati da salvare nella tabella.

DESCRIZIONE

Le funzioni *blk_cache_read*() e *blk_cache_save*() si occupano di trovare un blocco già salvato in precedenza nella tabella dei blocchi, o di salvarne uno nuovo o di aggiornarne il contenuto. Le funzioni restituiscono il puntatore al blocco trovato o memorizzato, oppure il puntatore nullo in caso di fallimento dell'operazione.

Quando si legge un blocco, lo si ottiene solo se viene trovata la corrispondenza con il numero di dispositivo (riferito all'unità complessiva) e al numero del blocco stesso. Quando si salva un blocco, viene prima cercato lo stesso blocco nella tabella, per aggiornarlo, ma se non c'è, viene riutilizzata una cella corrispondente a un vecchio blocco che non risulta usato da più tempo.

Una lettura con successo o il salvataggio di un blocco, implica l'attribuzione allo stesso di un'età pari a zero, modificando di conseguenza quella degli altri blocchi in modo coerente.

VALORE RESTITUITO

La funzione restituisce il puntatore all'area di memoria contenente il blocco di dati letto o scritto. Mentre il salvataggio avviene sempre con successo, la lettura può fallire e in tal caso si ottiene il puntatore nullo.

FILE SORGENTI

'kernel/blk.h' [94.2]

'kernel/blk/blk_cache_read.c' [94.2.4]

'kernel/blk/blk_cache_save.c' [94.2.5]

VEDERE ANCHE

dev_dm(9) [93.4.2], *blk_ata(9)* [93.3.1].

93.3.5 os32: *blk_cache_save(9)*

Vedere *blk_cache_read(9)* [93.3.4].

93.4 os32: dev(9)

«

Il file ‘kernel/dev.h’ [94.3] descrive ciò che serve per la gestione dei dispositivi. Tuttavia, la definizione dei numeri di dispositivo è contenuta nel file ‘lib/sys/os32.h’ [95.21], il quale viene incluso da ‘dev.h’.

Tabella 84.82. Classificazione dei dispositivi di os32.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_MEM	r/w	diret- to	Permette l’accesso alla memo- ria, in modo indiscriminato; tut- tavia, solo al kernel è permessa la scrittura.
DEV_NULL	r/w	nes- suno	Consente la lettura e la scrittura, ma non si legge e non si scrive alcunché.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_PORT	r/w	se- quen- ziale	Consente di leggere e scrivere da o verso una porta di I/O, individuata attraverso l'indirizzo di accesso (l'indirizzo, o meglio lo scostamento, viene trattato come la porta a cui si vuole accedere). Tuttavia, la dimensione dell'informazione da trasferire è valida solo se si tratta di uno o di due byte: per la dimensione di un byte si usano le funzioni <i>in_8()</i> e <i>out_8()</i> ; per due byte si usano le funzioni <i>in_16()</i> e <i>out_16()</i> . Per dimensioni differenti la lettura o la scrittura non ha effetto.
DEV_ZERO	r	se- quen- ziale	Consente solo la lettura di valori a zero (zero inteso in senso binario).
DEV_TTY	r/w	se- quen- ziale	Rappresenta il terminale virtuale del processo attivo.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_DM <i>m n</i>	r/w	diret- to	Rappresenta la partizione <i>n</i> dell'unità di memorizzazione <i>m</i> . La prima unità PATA disponibile ottiene il dispositivo <i>DEV_DM00</i> , la seconda il numero <i>DEV_DM10</i> , ecc.
DEV_KMEM_PS	r	diret- to	Rappresenta la tabella contenente le informazioni sui processi. L'indirizzo di accesso indica il numero del processo di partenza; la dimensione da leggere dovrebbe essere abbastanza grande da contenere un processo, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
DEV_KMEM_MMP	r	se- quen- ziale	Rappresenta la mappa della memoria, alla quale si può accedere solo dal suo principio. In pratica, l'indirizzo di accesso viene ignorato, mentre conta solo la quantità di byte richiesta.

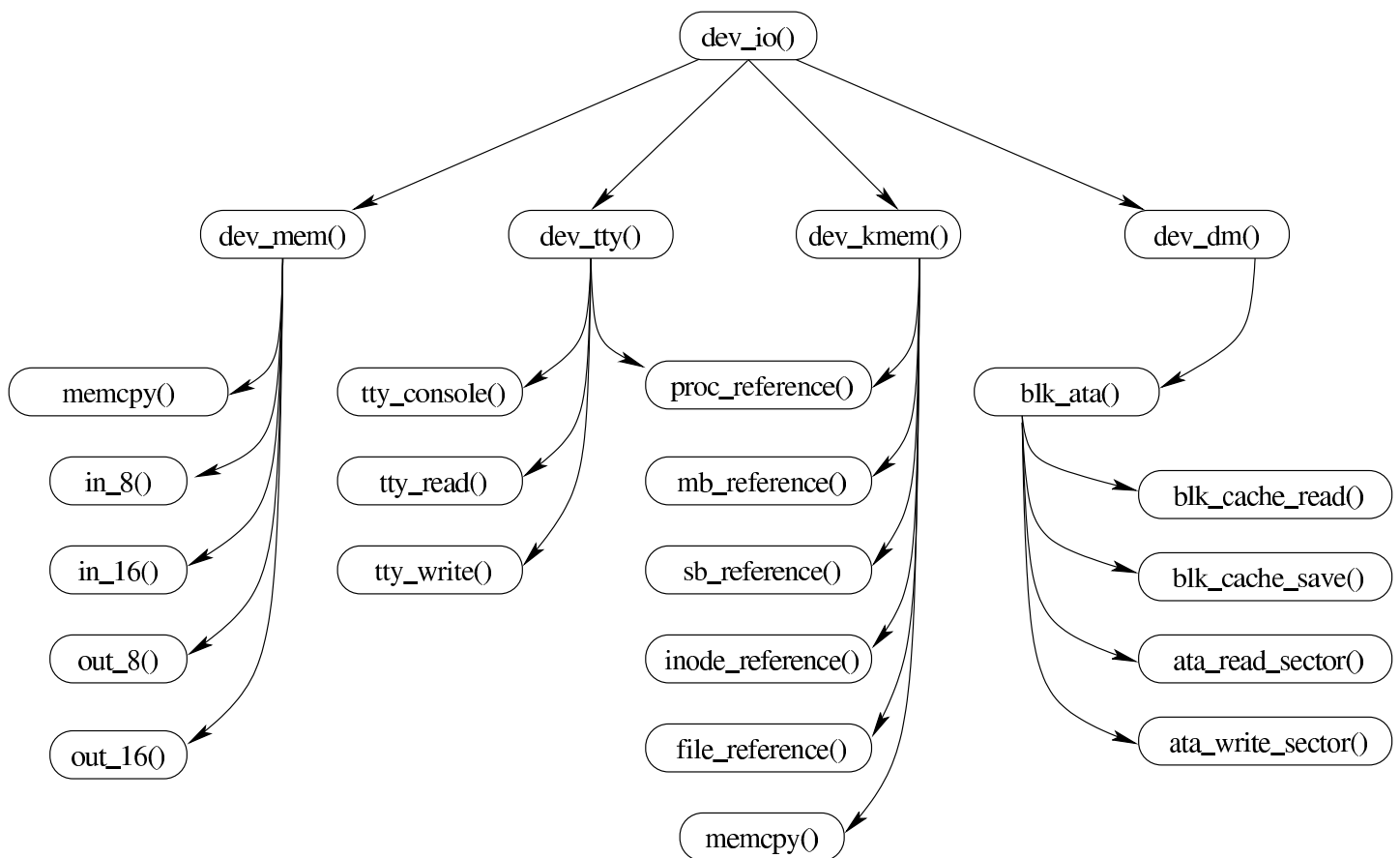
Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_KMEM_SB	r	diret- to	Rappresenta la tabella dei su- per blocchi (per la gestione delle unità di memorizzazio- ne). L'indirizzo di accesso ser- ve a individuare il super blocco; la dimensione richiesta dovb- be essere abbastanza grande da contenere un super blocco, ma anche richiedendo una dimen- sione maggiore, se ne legge uno solo.
DEV_KMEM_INODE	r	diret- to	Rappresenta la tabella degli ino- de (per la gestione delle unità di memorizzazione). L'indiriz- zo di accesso serve a individua- re l'inode; la dimensione richie- sta dovrebbe essere abbastanza grande da contenere un inode, ma anche richiedendo una di- mensione maggiore, se ne legge uno solo.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_KMEM_FILE	r	diret- to	Rappresenta la tabella dei file (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare il file; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di un file, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
DEV_KMEM_ARP	r	diret- to	Rappresenta la tabella ARP (per la trasformazione degli indirizzi IPv4 in indirizzi Ethernet). L'indirizzo di accesso serve a individuare la voce; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di una voce, ma anche richiedendo una dimensione maggiore, se ne legge una sola.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_KMEM_NET	r	diret- to	Rappresenta la tabella delle in- terfacce di rete. L'indirizzo di accesso serve a individuare la voce della tabella; la dimensio- ne richiesta dovrebbe essere ab- bastanza grande da contenere le informazioni di una voce, ma anche richiedendo una dimen- sione maggiore, se ne legge una sola.
DEV_KMEM_ROUTE	r	diret- to	Rappresenta la tabella degli in- stradamenti IPv4. L'indirizzo di accesso serve a individuare la voce della tabella; la dimensio- ne richiesta dovrebbe essere ab- bastanza grande da contenere le informazioni di una voce, ma anche richiedendo una dimen- sione maggiore, se ne legge una sola.
DEV_CONSOLE	r/w	se- quen- ziale	Legge o scrive relativamente alla console attiva la quanti- tà di byte richiesta, ignorando l'indirizzo di accesso.

Dispositivo	Let- tura e scrit- tura r/w	Ac- cesso diret- to o se- quen- ziale	Annotazioni
DEV_CONSOLE n	r/w	se- quen- ziale	Legge o scrive relativamente al- la console n la quantità di byte richiesta, ignorando l'indirizzo di accesso.

Figura 84.80. Interdipendenza tra la funzione *dev_io()* e le altre. I collegamenti con le funzioni *major()* e *minor()* sono omesse.



93.4.1 os32: dev_io(9)

**NOME**

‘**dev_io**’ - interfaccia di accesso ai dispositivi

SINTASSI

```
<kernel/dev.h>
ssize_t dev_io (pid_t pid, dev_t device, int rw, off_t offset,
               void *buffer, size_t size, int *eof);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Il numero del processo elaborativo per conto del quale si agisce.
dev_t <i>device</i>	Dispositivo, in forma numerica.
int <i>rw</i>	Può assumere i valori ‘ DEV_READ ’ o ‘ DEV_WRITE ’, per richiedere rispettivamente un accesso in lettura oppure in scrittura.
off_t <i>offset</i>	Posizione per l’accesso al dispositivo.
void * <i>buffer</i>	Memoria tampone, per la lettura o la scrittura.
size_t <i>size</i>	Quantità di byte da leggere o da scrivere.
int * <i>eof</i>	Puntatore a una variabile in cui annotare, eventualmente, il raggiungimento della fine del file.

DESCRIZIONE

La funzione *dev_io()* è un’interfaccia generale per l’accesso ai dispositivi gestiti da os32.

VALORE RESTITUITO

La funzione restituisce la quantità di byte letti o scritti effettivamente. In caso di errore restituisce il valore `-1` e aggiorna la variabile *errno* del kernel.

ERRORI

Valore di <i>errno</i>	Significato
ENODEV	Il numero del dispositivo non è valido.
EIO	Errore di input-output.

FILE SORGENTI

‘kernel/dev.h’ [94.3]

‘kernel/dev/dev_io.c’ [94.3.3]

VEDERE ANCHE

dev_dm(9) [93.4.2], *dev_kmem(9)* [93.4.4], *dev_mem(9)* [93.4.5], *dev_tty(9)* [93.4.6].

93.4.2 os32: dev_dm(9)

«

NOME

‘**dev_dm**’ - interfaccia di accesso alle unità di memorizzazione di massa

SINTASSI

```
<kernel/dev.h>
ssize_t dev_dm (pid_t pid, dev_t device, int rw, off_t offset,
                void *buffer, size_t size, int *eof);
```


DESCRIZIONE

La funzione *dev_dm()* consente di accedere alle unità di memorizzazione di massa, che per os32 si riducono alle sole unità PATA.

Per il significato degli argomenti, il valore restituito e gli eventuali errori, si veda *dev_io(9)* [93.4.1].

FILE SORGENTI

‘kernel/dev.h’ [94.3]

‘kernel/dev/dev_io.c’ [94.3.3]

‘kernel/dev/dev_dm.c’ [94.3.2]

93.4.3 os32: dev_ata(9)



NOME

‘**dev_ata**’ - interfaccia di accesso alle unità di memorizzazione di massa PATA

SINTASSI

```
<kernel/dev.h>
ssize_t dev_ata (pid_t pid, dev_t device, int rw, off_t offset,
                void *buffer, size_t size, int *eof);
```

DESCRIZIONE

La funzione *dev_ata()* consente di accedere alle unità di memorizzazione di massa PATA, traducendo le operazioni da flusso di byte a blocchi. Questa funzione viene usata da *dev_dm()* e si avvale a sua volta di *blk_ata()*.

Per il significato degli argomenti, il valore restituito e gli eventuali errori, si veda *dev_io(9)* [93.4.1].

FILE SORGENTI

‘kernel/dev.h’ [94.3]

‘kernel/dev/dev_io.c’ [94.3.3]

‘kernel/dev/dev_dm.c’ [94.3.2]

‘kernel/dev/dev_ata.c’ [94.3.1]

93.4.4 os32: dev_kmem(9)

«

NOME

‘**dev_kmem**’ - interfaccia di accesso alle tabelle di dati del kernel, rappresentate in memoria

SINTASSI

```
<kernel/dev.h>
ssize_t dev_kmem (pid_t pid, dev_t device, int rw,
                  off_t offset, void *buffer,
                  size_t size, int *eof);
```

DESCRIZIONE

La funzione *dev_kmem()* consente di accedere, solo in lettura, alle porzioni di memoria che il kernel utilizza per rappresentare alcune tabelle importanti. Per poter interpretare ciò che si ottiene occorre riprodurre la struttura di un elemento della tabella a cui si è interessati, pertanto occorre incorporare il file di intestazione del kernel che la descrive.

Dispositivo	Tabella a cui si riferisce
DEV_KMEM_PS	Si accede alla tabella dei processi, all'elemento rappresentato da <i>offset: proc_table[offset]</i> .
DEV_KMEM_MMP	Si accede alla mappa della memoria, ovvero la tabella <i>mb_table[]</i> , senza considerare il valore di <i>offset</i> .
DEV_KMEM_SB	Si accede alla tabella dei super blocchi, all'elemento rappresentato da <i>offset: sb_table[offset]</i> .
DEV_KMEM_INODE	Si accede alla tabella degli inode, all'elemento rappresentato da <i>offset: inode_table[offset]</i> .
DEV_KMEM_FILE	Si accede alla tabella dei file di sistema, all'elemento rappresentato da <i>offset: file_table[offset]</i> .

Per il significato degli argomenti della chiamata, per interpretare il valore restituito e gli eventuali errori, si veda *dev_io(9)* [93.4.1].

FILE SORGENTI

'kernel/dev.h' [94.3]

'kernel/dev/dev_io.c' [94.3.3]

'kernel/dev/dev_kmem.c' [94.3.4]

93.4.5 os32: dev_mem(9)

NOME

'**dev_mem**' - interfaccia di accesso alla memoria, in modo indiscriminato

SINTASSI

```
<kernel/fs.h>
ssize_t dev_mem (pid_t pid, dev_t device, int rw,
                 off_t offset, void *buffer,
                 size_t size, int *eof);
```

DESCRIZIONE

La funzione *dev_mem()* consente di accedere, in lettura e in scrittura alla memoria e alle porte di input-output.

Dispositivo	Descrizione
DEV_MEM	Si tratta della memoria centrale, complessiva, dove il valore di <i>offset</i> rappresenta l'indirizzo efficace (complessivo) a partire da zero.
DEV_NULL	Si tratta di ciò che realizza il file di dispositivo tradizionale <code>/dev/null</code> : la scrittura si perde semplicemente e la lettura non dà alcunché.
DEV_ZERO	Si tratta di ciò che realizza il file di dispositivo tradizionale <code>/dev/zero</code> : la scrittura si perde semplicemente e la lettura produce byte a zero (zero binario).
DEV_PORT	Consente l'accesso alle porte di input-output. La dimensione rappresentata da <i>size</i> può essere solo pari a uno o due: una dimensione pari a uno richiede di comunicare un solo byte con una certa porta; una dimensione pari a due richiede la comunicazione di un valore a 16 bit. Il valore di <i>offset</i> serve a individuare la porta di input-output con cui si intende comunicare (leggere o scrivere un valore).

Per quanto non viene descritto qui, si veda `dev_io(9)` [93.4.1].

FILE SORGENTI

'kernel/dev.h' [94.3]

'kernel/dev/dev_io.c' [94.3.3]

'kernel/dev/dev_mem.c' [94.3.5]

93.4.6 os32: dev_tty(9)

<<

NOME

‘**dev_tty**’ - interfaccia di accesso alla console

SINTASSI

```
<kernel/dev.h>
ssize_t dev_tty (pid_t pid, dev_t device, int rw, off_t offset,
                 void *buffer, size_t size, int *eof);
```

DESCRIZIONE

La funzione *dev_tty()* consente di accedere, in lettura e in scrittura, a una console virtuale, scelta in base al numero del dispositivo.

Quando la lettura richiede l’attesa per l’inserimento da tastiera, se il processo elaborativo *pid* non è il kernel, allora viene messo in pausa, in attesa di un evento legato al terminale.

Il sistema di gestione del terminale è molto povero con os32, in quanto il terminale può funzionare soltanto in modalità canonica; in altri termini, si può interagire soltanto come se si trattasse della telescrivente tradizionale.

Per quanto non viene descritto qui, si veda *dev_io(9)* [93.4.1].

FILE SORGENTI

‘kernel/dev.h’ [94.3]

‘kernel/dev/dev_io.c’ [94.3.3]

‘kernel/dev/dev_tty.c’ [94.3.6]

93.5 os32: dm(9)

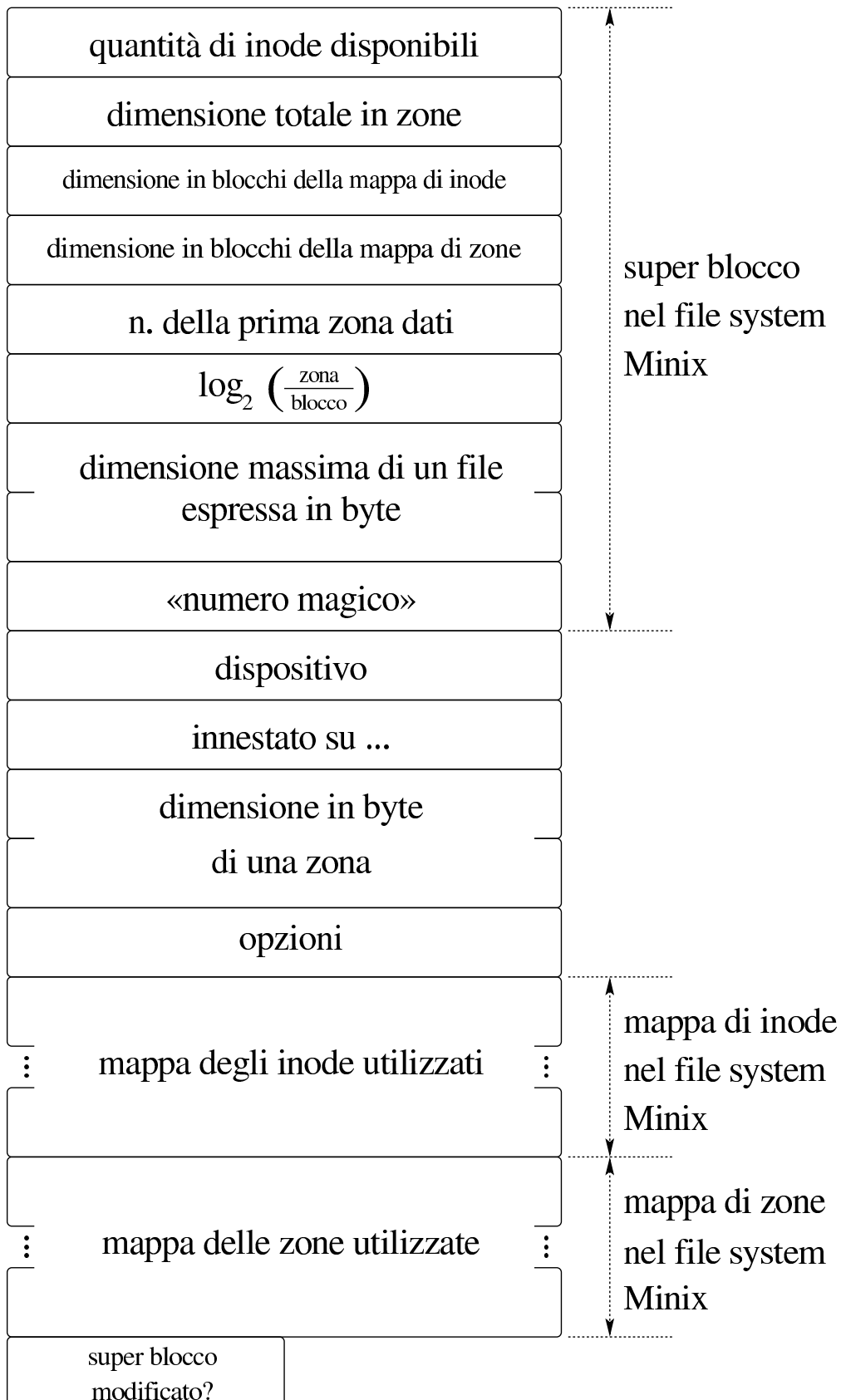
Il file ‘kernel/dm.h’ [94.4] descrive ciò che serve per la gestione delle unità di memorizzazione, in generale. È disponibile soltanto la funzione *dm_init()* per la predisposizione iniziale della tabella *dm_table[]*. «

93.6 os32: fs(9)

Il file ‘kernel/fs.h’ [94.5] descrive ciò che serve per la gestione del file system, che per os32 corrisponde al tipo Minix 1, assieme ai socket, essendo questi assimilati ai descrittori di file. «

La gestione del file system e dei socket, a livello complessivo di sistema, è suddivisa in quattro aspetti principali: super blocco, inode, file e socket. Per ognuno di questi è prevista una tabella (di super blocchi, di inode, di file e di socket). Seguono delle figure e i listati che descrivono l’organizzazione di queste tabelle.

Figura 84.95. Struttura del tipo 'sb_t', corrispondente agli elementi dell'array *sb_table[]*.

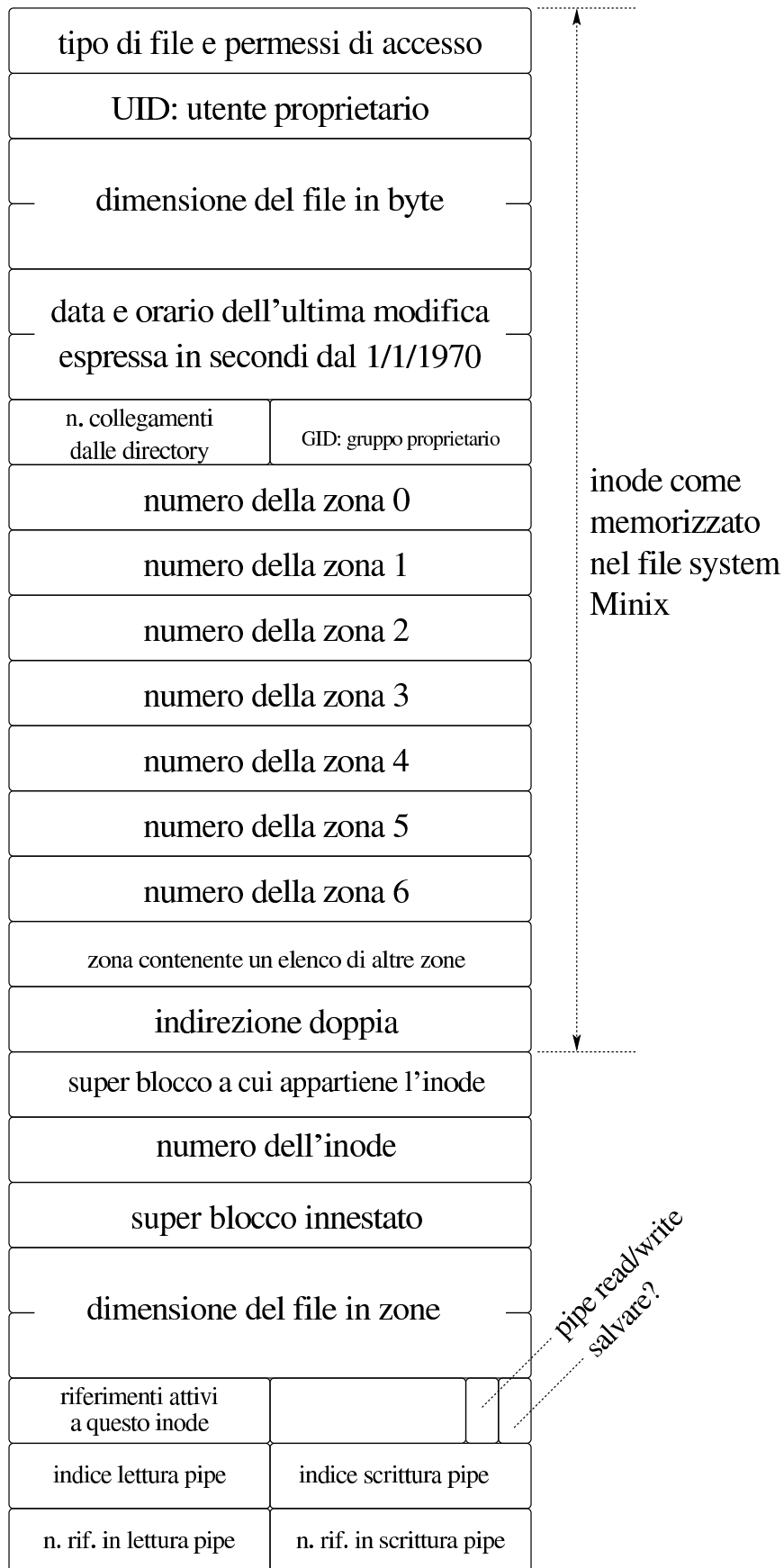


Listato 84.96. Struttura del tipo `'sb_t'`, corrispondente agli elementi dell'array `sb_table[]`.

```
typedef struct sb          sb_t;

struct sb {
    uint16_t  inodes;
    uint16_t  zones;
    uint16_t  map_inode_blocks;
    uint16_t  map_zone_blocks;
    uint16_t  first_data_zone;
    uint16_t  log2_size_zone;
    uint32_t  max_file_size;
    uint16_t  magic_number;
    //-----
    dev_t     device;
    inode_t   *inode_mounted_on;
    blksize_t blksize;
    int       options;
    uint16_t  map_inode[SB_MAP_INODE_SIZE];
    uint16_t  map_zone[SB_MAP_ZONE_SIZE];
    char      changed;
};
```

Figura 84.100. Struttura del tipo 'inode_t', corrispondente agli elementi dell'array *inode_table[]*.

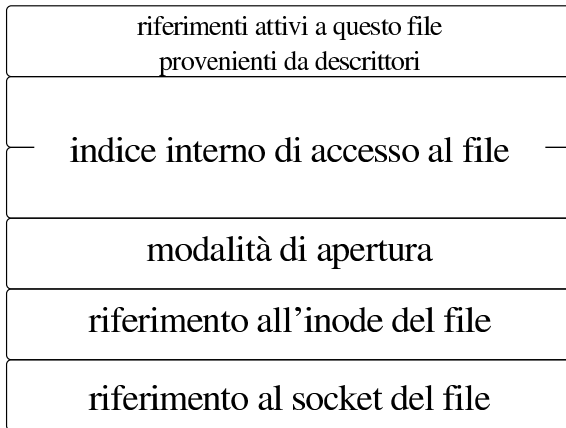


Listato 84.101. Struttura del tipo `'inode_t'`, corrispondente agli elementi dell'array `inode_table[]`.

```
typedef struct inode      inode_t;

struct inode {
    mode_t      mode;
    uid_t      uid;
    ssize_t     size;
    time_t     time;
    uint8_t    gid;
    uint8_t    links;
    zno_t      direct[7];
    zno_t      indirect1;
    zno_t      indirect2;
    //-----
    sb_t       *sb;
    ino_t      ino;
    sb_t       *sb_attached;
    blkcnt_t   blkcnt;
    unsigned char references;
    char       changed   : 1,
            pipe_dir   : 1;
    unsigned char pipe_off_read;
    unsigned char pipe_off_write;
    unsigned char pipe_ref_read;
    unsigned char pipe_ref_write;
};
```

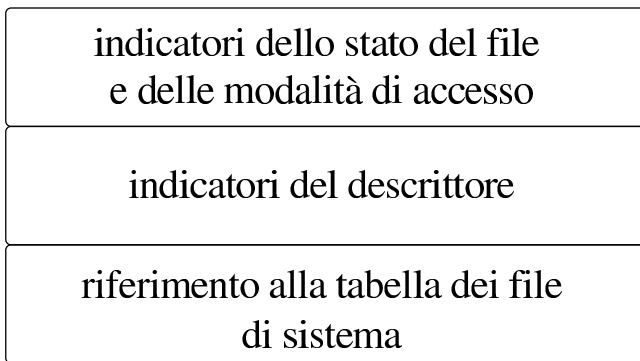
Figura 84.108. Struttura del tipo `'file_t'`, corrispondente agli elementi dell'array `file_table[]`.



```
typedef struct file file_t;

struct file {
    int         references;
    off_t       offset;
    int         oflags;
    inode_t     *inode;
    sock_t      *sock;
};
```

Figura 84.111. Struttura del tipo `'fd_t'`, con cui si costituiscono gli elementi delle tabelle dei descrittori di file, una per ogni processo.



```
typedef struct fd fd_t;
struct fd {
    int         fl_flags;
    int         fd_flags;
    file_t      *file;
};
```

Listato 84.131. Struttura di ogni elemento della tabella dei socket.

```
struct sock
{
    int         family;
    int         type;
    int         protocol;
    h_addr_t    laddr;           // indirizzo locale
    h_port_t    lport;          // porta locale
    h_addr_t    raddr;          // indirizzo remoto
    h_port_t    rport;          // porta remota
};
```

```

struct {
    clock_t    clock[IP_MAX_PACKETS];
} read;
uint8_t      active      : 1, // socket utilizzato?
            unreach_net  : 1, // rete irraggiungibile?
            unreach_host: 1, // destinazione
                                // irraggiungibile?
            unreach_prot: 1, // protocollo
                                // irraggiungibile?
            unreach_port: 1; // porta irraggiungibile?

struct
{
    uint16_t   conn      : 4, // stato della connessione
            can_write   : 1, // si può scrivere in
                                // `send_data[]'?
            can_read    : 1, // si può leggere a
                                // partire da
                                // `*recv_index'?
            can_send    : 1, // si possono inviare
                                // dati?
            can_recv    : 1, // si possono ricevere
                                // dati?
            send_closed : 1, // il canale di
                                // trasmissione
                                // è chiuso?
            recv_closed : 1; // il canale di ricezione
                                // è chiuso?

    //
    uint32_t   lsq[16]; // array della sequenza locale
    uint32_t   lsq_ack; // numero di sequenza in attesa
                                // di conferma
    uint32_t   rsq[16]; // array della sequenza remota
    uint8_t    lsqi     : 4, // indice dell'array della
                                // sequenza locale

```

```

        rsqi          : 4; // indice dell'array della
                        // sequenza remota

//
clock_t    clock;      // istante dell'ultima
                        // trasmissione

//
uint8_t    send_data[TCP_MSS - sizeof (struct tcphdr)];
                        // dati da trasmettere
size_t     send_size;  // dimensione dei dati da
                        // trasmettere

int        send_flags;
uint8_t    recv_data[TCP_MAX_DATA_SIZE];
                        // dati ricevuti
size_t     recv_size;  // dimensione dei dati
                        // ricevuti
uint8_t    *recv_index; // indice per la lettura dei
                        // dati ricevuti
pid_t     listen_pid;  // processo in ascolto della
                        // porta locale, in attesa di
                        // connessioni
int        listen_max; // numero massimo di richieste
                        // di connessione accettabili
int        listen_queue[SOCK_MAX_QUEUE];
                        // descrittori di connessioni
                        // realizzate

} tcp;
};

```

93.6.1 os32: fd_dup(9)

«

NOME

‘**fd_dup**’ - duplicazione di un descrittore di file

SINTASSI

```
<kernel/fs.h>
int fd_dup (pid_t pid, int fdn_old, int fdn_min);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Il numero del processo elaborativo per conto del quale si agisce.
int <i>fdn_old</i>	Numero del descrittore di file da duplicare.
int <i>fdn_min</i>	Primo numero di descrittore da usare per la copia.

DESCRIZIONE

La funzione *fd_dup()* duplica un descrittore, nel senso che sdoppia l'accesso a un file in due descrittori, cercando un descrittore libero a cominciare da *fdn_min* e continuando progressivamente, fino al primo disponibile. Il descrittore ottenuto dalla copia, viene privato dell'indicatore 'FD_CLOEXEC', ammesso che nel descrittore originale ci fosse.

Questa funzione viene usata da *s_dup(9)* [93.12] e da *s_fcntl(9)* [93.12], per la duplicazione di un descrittore.

VALORE RESTITUITO

La funzione restituisce il numero del descrittore prodotto dalla duplicazione. In caso di errore, invece, restituisce il valore -1 , aggiornando la variabile *errno* del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il valore di <i>fdn_min</i> è impossibile.
EBADF	Il valore di <i>fdn_old</i> non è valido.
EMFILE	Non è possibile allocare un nuovo descrittore.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/fd_dup.c’ [94.5.1]

VEDERE ANCHE

dup(2) [87.12], *dup2(2)* [87.12], *sysroutine(9)* [93.20.28],
proc_reference(9) [93.20.5].

93.6.2 os32: fd_reference(9)

«

NOME

‘**fd_reference**’ - riferimento a un elemento della tabella dei descrittori

SINTASSI

```
<kernel/fs.h>
fd_t *fd_reference (pid_t pid, int *fdn);
```


ARGOMENTI

Argomento	Descrizione
<code>pid_t pid</code>	Il numero del processo elaborativo per conto del quale si agisce.
<code>int fdn</code>	Il numero del descrittore di file.

DESCRIZIONE

La funzione *fd_reference()* restituisce il puntatore all'elemento della tabella dei descrittori, corrispondente al processo e al numero di descrittore specificati. Se però viene fornito un numero di descrittore negativo, si ottiene il puntatore al primo elemento che risulta libero nella tabella.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella dei descrittori, oppure il puntatore nullo in caso di errore, ma **senza aggiornare** la variabile *errno* del kernel. Infatti, l'unico errore che può verificarsi consiste nel non poter trovare il descrittore richiesto.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/fd_reference.c' [94.5.2]

VEDERE ANCHE

file_reference(9) [93.6.5], *inode_reference(9)* [93.6.21],
sb_reference(9) [93.6.29], *proc_reference(9)* [93.20.5].

93.6.3 os32: fs_init(9)

<<

NOME

‘**fs_init**’ - inizializzazione delle tabelle relative alla gestione del file system

SINTASSI

```
<kernel/fs.h>
void fs_init (void);
```

DESCRIZIONE

La funzione *fs_init()* azzerava il contenuto delle tabelle dei super blocchi, degli inode e dei file. Questa funzione viene usata esclusivamente da *kmain(9)* [93.13], per predisporre le tabelle di gestione del file system.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/fs_init.c’ [94.5.6]

VEDERE ANCHE

fd_reference(9) [93.6.2], *inode_reference(9)* [93.6.21],
sb_reference(9) [93.6.29].

93.6.4 os32: file_pipe_make(9)

<<

NOME

‘**file_pipe_make**’ - creazione di una voce relativa a un condotto, nella tabella dei file di sistema

SINTASSI

```
<kernel/fs.h>
file_t *file_pipe_make (void);
```

DESCRIZIONE

La funzione *file_pipe_make()* produce una voce nella tabella dei file di sistema, relativa a un condotto (*pipe*).

Per ottenere questo risultato occorre coinvolgere anche la funzione *inode_pipe_make(9)* [93.6.16], la quale si occupa di predisporre un inode, privo però di un collegamento a un file vero e proprio.

Questa funzione viene usata esclusivamente da *s_pipe(9)* [93.12], per la realizzazione della chiamata di sistema *pipe(2)* [87.38].

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella dei file di sistema, oppure il puntatore nullo in caso di errore, aggiornando la variabile *errno* del kernel.

ERRORI

Valore di <i>errno</i>	Significato
ENFILE	Non è possibile allocare un altro file di sistema.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/file_pipe_make.c’ [94.5.3]

VEDERE ANCHE

inode_pipe_make(9) [93.6.16], *file_reference*(9) [93.6.5],
inode_put(9) [93.6.20].

93.6.5 os32: *file_reference*(9)

<<

NOME

‘**file_reference**’ - riferimento a un elemento della tabella dei file di sistema

SINTASSI

```
<kernel/fs.h>
file_t *file_reference (int fno);
```

ARGOMENTI

Argomento	Descrizione
int <i>fno</i>	Il numero della voce della tabella dei file, a partire da zero.

DESCRIZIONE

La funzione *file_reference*() restituisce il puntatore all'elemento della tabella dei file di sistema, corrispondente al numero indicato come argomento. Se però tale numero fosse negativo, viene restituito il puntatore al primo elemento libero.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella dei file di sistema, oppure il puntatore nullo in caso di errore, ma **senza aggiornare** la variabile *errno* del kernel. Infatti, l'unico errore che può verificarsi consiste nel non poter trovare la voce richiesta.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/fs_public.c’ [94.5.7]

‘kernel/fs/file_reference.c’ [94.5.4]

VEDERE ANCHE

fd_reference(9) [93.6.2], *inode_reference(9)* [93.6.21],
sb_reference(9) [93.6.29], *proc_reference(9)* [93.20.5].

93.6.6 os32: file_stdio_dev_make(9)



NOME

‘**file_stdio_dev_make**’ - creazione di una voce relativa a un dispositivo di input-output standard, nella tabella dei file di sistema

SINTASSI

```
<kernel/fs.h>
file_t *file_stdio_dev_make (dev_t device, mode_t mode,
                             int oflags);
```

ARGOMENTI

Argomento	Descrizione
dev_t <i>device</i>	Il numero del dispositivo da usare per l’input o l’output.
mode_t <i>mode</i>	Tipo di file di dispositivo: è praticamente obbligatorio l’uso di ‘ S_IFCHR ’.
int <i>oflags</i>	Modalità di accesso: ‘ O_RDONLY ’ oppure ‘ O_WRONLY ’.

DESCRIZIONE

La funzione *file_stdio_dev_make()* produce una voce nella tabella dei file di sistema, relativa a un dispositivo di input-output, da usare come flusso standard. In altri termini, serve per creare le voci della tabella dei file, relative a standard input, standard output e standard error.

Per ottenere questo risultato occorre coinvolgere anche la funzione *inode_stdio_dev_make(9)* [93.6.23], la quale si occupa di predisporre un inode, privo però di un collegamento a un file vero e proprio.

Questa funzione viene usata esclusivamente da *proc_sys_exec(9)* [93.20.21], per attribuire standard input, standard output e standard error, che non fossero già disponibili.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella dei file di sistema, oppure il puntatore nullo in caso di errore, aggiornando la variabile *errno* del kernel.

ERRORI

Valore di <i>errno</i>	Significato
ENFILE	Non è possibile allocare un altro file di sistema.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/file_stdio_dev_make.c' [94.5.5]

VEDERE ANCHE

proc_sys_exec(9) [93.20.21], *inode_stdio_dev_make(9)* [93.6.23], *file_reference(9)* [93.6.5], *inode_put(9)* [93.6.20].

93.6.7 os32: *inode_alloc(9)*



NOME

‘**inode_alloc**’ - allocazione di un inode

SINTASSI

```
<kernel/fs.h>
inode_t *inode_alloc (dev_t device, mode_t mode, uid_t uid,
                      gid_t gid);
```

ARGOMENTI

Argomento	Descrizione
dev_t <i>device</i>	Il numero del dispositivo in cui si trova il file system dove allocare l'inode.
mode_t <i>mode</i>	Tipo di file e modalità dei permessi da associare all'inode.
uid_t <i>uid</i>	Utente proprietario dell'inode.
gid_t <i>gid</i>	Gruppo proprietario dell'inode.

DESCRIZIONE

La funzione *inode_alloc()* cerca un inode libero nel file system del dispositivo indicato, quindi lo alloca (lo segna come utilizzato) e lo modifica aggiornando il tipo e la modalità dei permessi, oltre al proprietario del file. Se la funzione riesce nel suo intento,

restituisce il puntatore all'inode in memoria, il quale rimane così aperto e disponibile per ulteriori elaborazioni.

Questa funzione viene usata esclusivamente da `path_inode_link(9)` [93.6.42], per la creazione di un nuovo file.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella degli inode di sistema, oppure il puntatore nullo in caso di errore, aggiornando la variabile *errno* del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il valore fornito per il parametro <i>mode</i> non è ammissibile.
ENODEV	Il dispositivo non corrisponde ad alcuna voce della tabella dei super blocchi; per esempio, il file system cercato potrebbe non essere ancora stato innestato.
ENOSPC	Non è possibile allocare l'inode, per mancanza di spazio.
ENFILE	Non c'è spazio nella tabella degli inode.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/inode_alloc.c' [94.5.8]

VEDERE ANCHE

`path_inode_link(9)` [93.6.42], `sb_reference(9)` [93.6.29],
`inode_get(9)` [93.6.15], `inode_put(9)` [93.6.20],
`inode_truncate(9)` [93.6.24], `inode_save(9)` [93.6.22].

93.6.8 os32: inode_check(9)

**NOME**

‘**inode_check**’ - verifica delle caratteristiche di un inode

SINTASSI

```
<kernel/fs.h>
int inode_check (inode_t *inode, mode_t type, int perm,
                 uid_t uid, gid_t gid);
```

ARGOMENTI

Argomento	Descrizione
inode_t * <i>inode</i>	Puntatore a un elemento della tabella degli inode.
mode_t <i>type</i>	Tipo di file desiderato. Può trattarsi di ‘S_IFBLK’, ‘S_IFCHR’, ‘S_IFIFO’, ‘S_IFREG’, ‘S_IFDIR’, ‘S_IFLNK’, ‘S_IFSOCK’, come dichiarato nel file ‘lib/sys/stat.h’, tuttavia os32 gestisce solo file di dispositivo, file normali e directory.
int <i>perm</i>	Permessi richiesti dall’utente <i>uid</i> , rappresentati nei tre bit meno significativi.
uid_t <i>uid</i>	Utente nei confronti del quale vanno verificati i permessi di accesso.
gid_t <i>gid</i>	Gruppo di utenti nei confronti del quale vanno verificati i permessi di accesso.

DESCRIZIONE

La funzione *inode_check()* verifica che l’inode indicato sia di un certo tipo e abbia i permessi di accesso necessari a un certo utente e gruppo. Tali permessi vanno rappresentati utilizzando

solo gli ultimi tre bit (4 = lettura, 2 = scrittura, 1 = esecuzione o attraversamento) e si riferiscono alla richiesta di accesso all'inode, da parte dell'utente *uid* e del gruppo *gid*, tenendo conto del complesso dei permessi che li riguardano.

Nel parametro *type* è ammessa la sovrapposizione di più tipi validi.

Questa funzione viene usata in varie situazioni, internamente al kernel, per verificare il tipo o l'accessibilità di un file.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Le caratteristiche dell'inode sono compatibili con quanto richiesto.
-1	Le caratteristiche dell'inode non sono compatibili, oppure si è verificato un errore. In ogni caso si ottiene l'aggiornamento della variabile <i>errno</i> del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il valore di <i>inode</i> corrisponde a un puntatore nullo.
E_FILE_TYPE	Il tipo di file dell'inode non corrisponde a quanto richiesto.
EACCES	I permessi di accesso non sono compatibili con la richiesta.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/inode_check.c' [94.5.9]

93.6.9 os32: inode_dir_empty(9)

**NOME**

‘**inode_dir_empty**’ - verifica della presenza di contenuti in una directory

SINTASSI

```
<kernel/fs.h>
int inode_dir_empty (inode_t *inode);
```

ARGOMENTI

Argomento	Descrizione
inode_t * <i>inode</i>	Puntatore a un elemento della tabella degli inode.

DESCRIZIONE

La funzione *inode_dir_empty()* verifica che la directory, a cui si riferisce l’inode a cui punta *inode*, sia vuota.

VALORE RESTITUITO

Valore	Significato del valore restituito
1	<i>Vero</i> : si tratta di una directory vuota.
0	<i>Falso</i> : se è effettivamente una directory, questa non è vuota, altrimenti non è nemmeno una directory.

ERRORI

Dal momento che un risultato *Falso* non rappresenta necessariamente un errore, per verificare il contenuto della variabile *errno*, prima dell’uso della funzione occorre azzerarla.

Valore di <i>errno</i>	Significato
EINVAL	L'inode non riguarda una directory.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/inode_dir_empty.c' [94.5.10]

VEDERE ANCHE

inode_file_read(9) [93.6.10].

93.6.10 os32: *inode_file_read*(9)

<<

NOME

'*inode_file_read*' - lettura di un file rappresentato da un inode

SINTASSI

```
<kernel/fs.h>
ssize_t inode_file_read (inode_t *inode, off_t offset,
                        void *buffer, size_t count,
                        int *eof);
```

ARGOMENTI

Argomento	Descrizione
<code>inode_t *inode</code>	Puntatore a un elemento della tabella degli inode, che rappresenta il file da leggere.
<code>off_t offset</code>	Posizione, riferita all'inizio del file, a partire dalla quale eseguire la lettura.
<code>void *buffer</code>	Puntatore all'area di memoria in cui scrivere ciò che si ottiene dalla lettura del file.
<code>size_t count</code>	Quantità massima di byte da leggere.
<code>int *eof</code>	Puntatore a un indicatore di fine file, da aggiornare (purché sia un puntatore valido) in base all'esito della lettura.

DESCRIZIONE

La funzione `inode_file_read()` legge il contenuto del file a cui si riferisce l'inode `inode` e se il puntatore `eof` è valido, aggiorna anche la variabile `*eof`.

Questa funzione si avvale a sua volta di `inode_fzones_read(9)` [93.6.13], per accedere ai contenuti del file, suddivisi in zone, secondo l'organizzazione del file system Minix 1.

VALORE RESTITUITO

La funzione restituisce la quantità di byte letti e resi effettivamente disponibili a partire da ciò a cui punta `buffer`. Se la variabile `var` è un puntatore valido, aggiorna anche il suo valore, azzerandolo se la lettura avviene in una posizione interna al file, oppure impostandolo a uno se la lettura richiesta è oltre la fine del file. Se invece si tenta una lettura con un valore di `offset` negativo,

ARGOMENTI

Argomento	Descrizione
<code>inode_t *inode</code>	Puntatore a un elemento della tabella degli inode, che rappresenta il file da scrivere.
<code>off_t offset</code>	Posizione, riferita all'inizio del file, a partire dalla quale eseguire la scrittura.
<code>void *buffer</code>	Puntatore all'area di memoria da cui trarre i dati da scrivere nel file.
<code>size_t count</code>	Quantità massima di byte da scrivere.

DESCRIZIONE

La funzione `inode_file_write()` scrive nel file rappresentato da *inode*, a partire dalla posizione *offset* (purché non sia un valore negativo), la quantità massima di byte indicati con *count*, ciò che si trova in memoria a partire da *buffer*.

Questa funzione si avvale a sua volta di `inode_fzones_read(9)` [93.6.13], per accedere ai contenuti del file, suddivisi in zone, secondo l'organizzazione del file system Minix 1, e di `zone_write(9)` [93.6.37], per la riscrittura delle zone relative.

Per os32, le operazioni di scrittura nel file system sono sincrone, senza alcun trattenimento in memoria (ovvero senza *cache*).

VALORE RESTITUITO

La funzione restituisce la quantità di byte scritti. La scrittura può avvenire oltre la fine del file, anche in modo discontinuo; tuttavia, non è ammissibile un valore di *offset* negativo.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	L'inode non è valido, oppure il valore di <i>offset</i> è negativo.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/inode_file_write.c' [94.5.12]

VEDERE ANCHE

inode_fzones_read(9) [93.6.13], *zone_write*(9) [93.6.37].

93.6.12 os32: *inode_free*(9)

«

NOME

'*inode_free*' - deallocazione di un inode

SINTASSI

```
<kernel/fs.h>
int inode_free (inode_t *inode);
```

ARGOMENTI

Argomento	Descrizione
<code>inode_t *inode</code>	Puntatore a un elemento della tabella degli inode.

DESCRIZIONE

La funzione *inode_free*() libera l'inode specificato attraverso il puntatore *inode*, rispetto al proprio super blocco. L'operazione

comporta semplicemente il fatto di indicare questo inode come libero, senza controlli per verificare se effettivamente non esistono più collegamenti nel file system che lo riguardano.

Questa funzione viene usata esclusivamente da *inode_put(9)* [93.6.20], per completare la cancellazione di un inode che non ha più collegamenti nel file system, quando non vi si fa più riferimento nel sistema in funzione.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dalla variabile <i>errno</i> del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	L'inode non è valido.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/inode_free.c' [94.5.13]

VEDERE ANCHE

inode_save(9) [93.6.22], *inode_alloc(9)* [93.6.7].

93.6.13 os32: inode_fzones_read(9)



NOME

'*inode_fzones_read*', '*inode_fzones_write*' - lettura e scrittura di zone relative al contenuto di un file

SINTASSI

```
<kernel/fs.h>
blkcnt_t inode_fzones_read (inode_t *inode, zno_t zone_start,
                             void *buffer, blkcnt_t blkcnt);
blkcnt_t inode_fzones_write (inode_t *inode, zno_t zone_start,
                              void *buffer, blkcnt_t blkcnt);
```

ARGOMENTI

Argomento	Descrizione
inode_t * <i>inode</i>	Puntatore a un elemento della tabella degli inode, con cui si individua il file da cui leggere o in cui scrivere.
zno_t <i>zone_start</i>	Il numero di zona, relativo al file, a partire dalla quale iniziare la lettura o la scrittura.
void * <i>buffer</i>	Il puntatore a un'area di memoria tampone, da usare per depositare i dati letti o per trarre i dati da scrivere.
blkcnt_t <i>blkcnt</i>	La quantità di zone da leggere o scrivere.

DESCRIZIONE

Le funzioni *inode_fzones_read()* e *inode_fzones_write()*, consentono di leggere e di scrivere un file, a zone intere (la zona è un multiplo del blocco, secondo la filosofia del file system Minix 1). Questa funzione vengono usate soltanto da *inode_file_read(9)* [93.6.10] e *inode_file_write(9)* [93.6.11], con le quali l'accesso ai file si semplifica a livello di byte.

VALORE RESTITUITO

Le due funzioni restituiscono la quantità di zone lette o scritte

effettivamente. Una quantità pari a zero potrebbe eventualmente rappresentare un errore, ma solo in alcuni casi. Per poterlo verificare, occorre azzerare la variabile *errno* prima di chiamare le funzioni, riservandosi di verificarne successivamente il valore.

ERRORI

Valore di <i>errno</i>	Significato
EIO	L'accesso alla zona richiesta non è potuto avvenire.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/inode_fzones_read.c' [94.5.14]

'kernel/fs/inode_fzones_write.c' [94.5.15]

VEDERE ANCHE

inode_file_read(9) [93.6.10], *inode_file_write(9)* [93.6.11],
zone_read(9) [93.6.37], *zone_write(9)* [93.6.37].

93.6.14 os32: *inode_fzones_write(9)*

Vedere *inode_fzones_read(9)* [93.6.13].

93.6.15 os32: *inode_get(9)*

NOME

'*inode_get*' - caricamento di un inode

SINTASSI

```
<kernel/fs.h>
inode_t *inode_get (dev_t device, ino_t ino);
```

ARGOMENTI

Argomento	Descrizione
<code>dev_t</code> <i>device</i>	Dispositivo riferito a un'unità di memorizzazione, dove cercare l'inode numero <i>ino</i> .
<code>ino_t</code> <i>ino</i>	Numero di inode, relativo al file system contenuto nell'unità <i>device</i> .

DESCRIZIONE

La funzione *inode_get()* consente di «aprire» un inode, fornendo il numero del dispositivo corrispondente all'unità di memorizzazione e il numero dell'inode del file system in essa contenuto. L'inode in questione potrebbe essere già stato aperto e quindi già disponibile in memoria nella tabella degli inode; in tal caso, la funzione si limita a incrementare il contatore dei riferimenti a tale inode, da parte del sistema in funzione, restituendo il puntatore all'elemento della tabella che lo contiene già. Se invece l'inode non è ancora presente nella tabella rispettiva, la funzione deve provvedere a caricarlo.

Se si richiede un inode non ancora disponibile, contenuto in un'unità di cui non è ancora stato caricato il super blocco nella tabella rispettiva, la funzione deve provvedere anche a questo procedimento.

VALORE RESTITUITO

La funzione restituisce il puntatore all'elemento della tabella degli inode che rappresenta l'inode aperto. Se però si presenta un problema, restituisce il puntatore nullo e aggiorna la variabile *errno* del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EUNKNOWN	Si tratta di un problema imprevisto e non meglio identificabile.
ENFILE	La tabella degli inode è già occupata completamente e non è possibile aprirne altri.
ENODEV	Il dispositivo richiesto non è valido.
ENOENT	Il numero di inode richiesto non esiste.
EIO	Errore nella lettura del file system.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/inode_get.c’ [94.5.16]

VEDERE ANCHE

offsetof(3) [88.88], *inode_put(9)* [93.6.20], *inode_reference(9)* [93.6.21], *sb_reference(9)* [93.6.29], *sb_inode_status(9)* [93.6.26], *dev_io(9)* [93.4.1].

93.6.16 os32: *inode_pipe_make(9)*

«

NOME

‘**inode_pipe_make**’ - creazione di una voce relativa a un condotto, nella tabella degli inode

SINTASSI

```
<kernel/fs.h>
inode_t *inode_pipe_make (void);
```

DESCRIZIONE

La funzione *inode_pipe_make()* produce una voce nella tabella degli inode, relativa a un condotto (*pipe*).

Questa funzione viene usata esclusivamente da *file_pipe_make(9)* [93.6.4], per creare una voce da usare condotto, nella tabella dei file.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella degli inode, oppure il puntatore nullo in caso di errore, aggiornando la variabile *errno* del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Gli argomenti della chiamata non sono validi.
ENFILE	Non è possibile allocare un altro inode.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/inode_pipe_make.c’ [94.5.17]

VEDERE ANCHE

file_pipe_make(9) [93.6.4], *inode_reference(9)* [93.6.21].

93.6.17 os32: *inode_pipe_read(9)*

«

NOME

‘*inode_pipe_read*’ - lettura di un condotto rappresentato da un inode

SINTASSI

```
<kernel/fs.h>
ssize_t inode_file_read (inode_t *inode, void *buffer,
                          size_t count, int *eof);
```

ARGOMENTI

Argomento	Descrizione
<code>inode_t *<i>inode</i></code>	Puntatore a un elemento della tabella degli inode, che rappresenta il condotto da leggere.
<code>void *<i>buffer</i></code>	Puntatore all'area di memoria in cui scrivere ciò che si ottiene dalla lettura del condotto.
<code>size_t <i>count</i></code>	Quantità massima di byte da leggere.
<code>int *<i>eof</i></code>	Puntatore a un indicatore di fine file, da aggiornare (purché sia un puntatore valido) in base all'esito della lettura.

DESCRIZIONE

La funzione `inode_pipe_read()` legge il contenuto del file a cui si riferisce l'inode `inode` e se il puntatore `eof` è valido, aggiorna anche la variabile `*eof`.

VALORE RESTITUITO

La funzione restituisce la quantità di byte letti e resi effettivamente disponibili a partire da ciò a cui punta `buffer`. Se la variabile `var` è un puntatore valido, aggiorna anche il suo valore, impostandolo a uno quando la lettura avviene mentre non ci sono più riferimenti in scrittura per il condotto.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	L'inode non è valido.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/inode_pipe_read.c' [94.5.18]

VEDERE ANCHE

inode_file_read(9) [93.6.10].

93.6.18 os32: *inode_pipe_write(9)*

«

NOME

'**inode_pipe_write**' - scrittura di un condotto rappresentato da un inode

SINTASSI

```
<kernel/fs.h>
ssize_t inode_pipe_write (inode_t *inode, void *buffer,
                          size_t count);
```


ARGOMENTI

Argomento	Descrizione
<code>inode_t *<i>inode</i></code>	Puntatore a un elemento della tabella degli inode che rappresenta il condotto in cui scrivere.
<code>void *<i>buffer</i></code>	Puntatore all'area di memoria da cui trarre i dati da scrivere nel condotto.
<code>size_t <i>count</i></code>	Quantità massima di byte da scrivere.

DESCRIZIONE

La funzione `inode_pipe_write()` scrive nel condotto rappresentato da *inode*, la quantità massima di byte indicati con *count*, ciò che si trova in memoria a partire da *buffer*.

VALORE RESTITUITO

La funzione restituisce la quantità di byte scritti.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	L'inode non è valido.

FILE SORGENTI

'kernel/fs.h' [[94.5](#)]

'kernel/fs/inode_pipe_write.c' [[94.5.19](#)]

VEDERE ANCHE

`inode_file_write(9)` [[93.6.11](#)].

93.6.19 os32: inode_print(9)

<<

NOME

‘**inode_print**’ - visualizzazione diagnostica della tabella degli inode

SINTASSI

```
<kernel/fs.h>
void inode_print (void);
```

DESCRIZIONE

La funzione *inode_print()* visualizza sinteticamente i contenuti della tabella degli inode, per fini diagnostici.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/inode_print.c’ [94.5.20]

VEDERE ANCHE

sb_print(9) [93.6.28], *zone_print(9)* [93.6.36].

93.6.20 os32: inode_put(9)

<<

NOME

‘**inode_put**’ - rilascio di un inode

SINTASSI

```
<kernel/fs.h>
int inode_put (inode_t *inode);
```

ARGOMENTI

Argomento	Descrizione
<code>inode_t *inode</code>	Puntatore a un elemento della tabella di inode.

DESCRIZIONE

La funzione *inode_put()* «chiude» un inode, riducendo il contatore degli accessi allo stesso. Tuttavia, se questo contatore, dopo il decremento, raggiunge lo zero, è necessario verificare se nel frattempo anche i collegamenti del file system si sono azzerati, perché in tal caso occorre anche rimuovere l'inode, nel senso di segnalarlo come libero per la creazione di un nuovo file. In ogni caso, le informazioni aggiornate dell'inode, ancora allocato o liberato, vengono memorizzate nel file system.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dalla variabile <i>errno</i> del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il puntatore <i>inode</i> non è valido.
EUNKNOWN	Si tratta di un problema imprevisto e non meglio identificabile.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/inode_put.c' [94.5.21]

VEDERE ANCHE

inode_truncate(9) [93.6.24], *inode_free*(9) [93.6.12],
inode_save(9) [93.6.22].

93.6.21 os32: *inode_reference*(9)

«

NOME

‘**inode_reference**’ - riferimento a un elemento della tabella di inode

SINTASSI

```
<kernel/fs.h>
inode_t *inode_reference (dev_t device, ino_t ino);
```

ARGOMENTI

Argomento	Descrizione
dev_t <i>device</i>	Dispositivo riferito a un'unità di memorizzazione, dove cercare l'inode numero <i>ino</i> .
ino_t <i>ino</i>	Numero di inode, relativo al file system contenuto nell'unità <i>device</i> .

DESCRIZIONE

La funzione *inode_reference*() cerca nella tabella degli inode la voce corrispondente ai dati forniti come argomenti, ovvero quella dell'inode numero *ino* del file system contenuto nel dispositivo *device*, restituendo il puntatore alla voce corrispondente. Tuttavia ci sono dei casi particolari:

- se il numero del dispositivo e quello dell'inode sono entrambi zero, viene restituito il puntatore all'inizio della tabella, ovvero al primo elemento della stessa;

- se il numero del dispositivo e quello dell'inode sono pari a un numero negativo (rispettivamente '`(dev_t) -1`' e '`(ino_t) -1`'), viene restituito il puntatore alla prima voce libera;
- se il numero del dispositivo è pari a zero e il numero dell'inode è pari a uno, si intende ricercare la voce dell'inode della directory radice del file system principale.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella degli inode, se la ricerca si compie con successo. In caso di problemi, invece, la funzione restituisce il puntatore nullo e aggiorna la variabile *errno* del kernel.

ERRORI

Valore di <i>errno</i>	Significato
<code>E_CANNOT_FIND_ROOT_DEVICE</code>	Nella tabella dei super blocchi non è possibile trovare il file system principale.
<code>E_CANNOT_FIND_ROOT_INODE</code>	Nella tabella degli inode non è possibile trovare la directory radice del file system principale.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/inode_reference.c' [94.5.22]

VEDERE ANCHE

sb_reference(9) [93.6.29], *file_reference*(9) [93.6.5],
proc_reference(9) [93.20.5].

93.6.22 os32: inode_save(9)

<<

NOME

‘**inode_save**’ - memorizzazione dei dati di un inode

SINTASSI

```
<kernel/fs.h>
int inode_save (inode_t *inode);
```

ARGOMENTI

Argomento	Descrizione
inode_t * <i>inode</i>	Puntatore a una voce della tabella degli inode.

DESCRIZIONE

La funzione *inode_save()* memorizza l’inode a cui si riferisce la voce **inode*, nel file system, ammesso che si tratti effettivamente di un inode relativo a un file system e che sia stato modificato dopo l’ultima memorizzazione precedente. In questo caso, la funzione, a sua volta, richiede la memorizzazione del super blocco.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dalla variabile <i>errno</i> del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il puntatore <i>inode</i> è nullo.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/inode_save.c’ [94.5.23]

VEDERE ANCHE

sb_save(9) [93.6.30], *dev_io(9)* [93.4.1].

93.6.23 os32: inode_stdio_dev_make(9)



NOME

‘**inode_stdio_dev_make**’ - creazione di una voce relativa a un dispositivo di input-output standard, nella tabella degli inode

SINTASSI

```
<kernel/fs.h>
inode_t *inode_stdio_dev_make (dev_t device, mode_t mode);
```

ARGOMENTI

Argomento	Descrizione
dev_t <i>device</i>	Il numero del dispositivo da usare per l’input o l’output.
mode_t <i>mode</i>	Tipo di file di dispositivo: è praticamente obbligatorio l’uso di ‘S_IFCHR’.

DESCRIZIONE

La funzione *inode_stdio_dev_make()* produce una voce nella tabella degli inode, relativa a un dispositivo di input-output, da usare come flusso standard. In altri termini, serve per creare le voci della tabella degli inode, relative a standard input, standard output e standard error.

Questa funzione viene usata esclusivamente da *file_stdio_dev_make(9)* [93.6.6], per creare una voce da usare come flusso standard di input o di output, nella tabella dei file.

VALORE RESTITUITO

La funzione restituisce il puntatore a un elemento della tabella degli inode, oppure il puntatore nullo in caso di errore, aggiornando la variabile *errno* del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Gli argomenti della chiamata non sono validi.
ENFILE	Non è possibile allocare un altro inode.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/inode_stdio_dev_make.c’ [94.5.24]

VEDERE ANCHE

file_stdio_dev_make(9) [93.6.6], *inode_reference(9)* [93.6.21].

93.6.24 os32: *inode_truncate(9)*

«

NOME

‘**inode_truncate**’ - troncamento del file a cui si riferisce un inode

SINTASSI

```
<kernel/fs.h>
int inode_truncate (inode_t *inode);
```

ARGOMENTI

Argomento	Descrizione
inode_t * <i>inode</i>	Puntatore a una voce della tabella di inode.

DESCRIZIONE

La funzione *inode_truncate()* richiede che il puntatore *inode* si riferisca a una voce della tabella degli inode, relativa a un file contenuto in un file system. Lo scopo della funzione è annullare il contenuto di tale file, trasformandolo in un file vuoto.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dalla variabile <i>errno</i> del kernel.

ERRORI

Allo stato attuale dello sviluppo della funzione, non ci sono controlli e non sono previsti errori.

FILE SORGENTI

‘kernel/fs.h’ [[94.5](#)]

‘kernel/fs/inode_truncate.c’ [[94.5.25](#)]

VEDERE ANCHE

zone_free(9) [[93.6.34](#)], *sb_save(9)* [[93.6.30](#)], *inode_save(9)* [[93.6.22](#)].

93.6.25 os32: inode_zone(9)

<<

NOME

‘**inode_zone**’ - traduzione del numero di zona relativo in un numero di zona assoluto

SINTASSI

```
<kernel/fs.h>
zno_t inode_zone (inode_t *inode, zno_t fzone, int write);
```

ARGOMENTI

Argomento	Descrizione
inode_t * <i>inode</i>	Puntatore a una voce della tabella di inode.
zno_t <i>fzone</i>	Numero di zona relativo al file dell'inode preso in considerazione.
int <i>write</i>	Valore da intendersi come <i>Vero</i> o <i>Falso</i> , con cui consentire o meno la creazione al volo di una zona mancante.

DESCRIZIONE

La funzione *inode_zone()* serve a tradurre il numero di una zona, inteso relativamente a un file, nel numero assoluto relativamente al file system in cui si trova. Tuttavia, un file può essere memorizzato effettivamente in modo discontinuo, ovvero con zone inesistenti nella sua parte centrale. Il contenuto di un file che non dispone effettivamente di zone allocate, corrisponde a un contenuto nullo dal punto di vista binario (zero binario), ma per la funzione, una zona assente comporta la restituzione di un valore nullo, perché nel file system non c'è. Pertanto, se l'argomento corrispondente al parametro *write* contiene un valore diverso da

zero, la funzione che non trova una zona, la alloca e quindi ne restituisce il numero.

VALORE RESTITUITO

La funzione restituisce il numero della zona che nel file system corrisponde a quella relativa richiesta per un certo file. Nel caso la zona non esista, perché non allocata, restituisce zero. Tuttavia, la zona zero di un file system Minix 1 esiste, ma contiene sostanzialmente le informazioni amministrative del super blocco, pertanto non può essere una traduzione valida di una zona di un file.

ERRORI

La funzione non prevede il verificarsi di errori.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/inode_zone.c’ [94.5.26]

VEDERE ANCHE

memset(3) [88.82], *zone_alloc(9)* [93.6.34], *zone_read(9)* [93.6.37], *zone_write(9)* [93.6.37].

93.6.26 os32: *sb_inode_status(9)*

«

NOME

‘**sb_inode_status**’, ‘**sb_zone_status**’ - verifica di utilizzazione attraverso il controllo delle mappe di inode e di zone

SINTASSI

```
<kernel/fs.h>
int sb_inode_status (sb_t *sb, ino_t ino);
int sb_zone_status (sb_t *sb, zno_t zone);
```

ARGOMENTI

Argomento	Descrizione
sb_t *sb	Puntatore a una voce della tabella dei super blocchi.
ino_t ino	Numero di inode.
zno_t ino	Numero di zona.

DESCRIZIONE

La funzione *sb_inode_status()* verifica che un certo inode, individuato per numero, risulti utilizzato nel file system a cui si riferisce il super blocco a cui punta il primo argomento.

La funzione *sb_zone_status()* verifica che una certa zona, individuato per numero, risulti utilizzata nel file system a cui si riferisce il super blocco a cui punta il primo argomento.

La funzione *sb_inode_status()* viene usata soltanto da *inode_get(9)* [93.6.15]; la funzione *sb_zone_status()* non viene usata affatto.

VALORE RESTITUITO

Valore	Significato
1	L'inode o la zona risultano utilizzati.
0	L'inode o la zona risultano liberi (allocabili).
-1	Errore: è stato richiesto un numero di inode o di zona pari a zero, oppure <i>sb</i> è un puntatore nullo.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	È stato richiesto un numero di inode o di zona pari a zero, oppure <i>sb</i> è un puntatore nullo.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/sb_inode_status.c’ [94.5.32]

‘kernel/fs/sb_zone_status.c’ [94.5.37]

VEDERE ANCHE

inode_alloc(9) [93.6.7], *zone_alloc(9)* [93.6.34].

93.6.27 os32: *sb_mount(9)*



NOME

‘**sb_mount**’ - innesto di un dispositivo di memorizzazione

SINTASSI

```
<kernel/fs.h>
sb_t *sb_mount (dev_t device, inode_t **inode_mnt,
                int options);
```

ARGOMENTI

Argomento	Descrizione
<code>dev_t <i>device</i></code>	Dispositivo da innestare.
<code>inode_t **<i>inode_mnt</i></code>	Puntatore di puntatore a una voce della tabella di inode. Il valore di <i>*inode_mnt</i> potrebbe essere un puntatore nullo.
<code>int <i>options</i></code>	Opzioni per l'innesto.

DESCRIZIONE

La funzione *sb_mount()* innesta il dispositivo rappresentato numericamente dal primo parametro, sulla directory corrispondente all'inode a cui punta, indirettamente, il secondo parametro, con le opzioni del terzo parametro.

Il secondo parametro è un puntatore di puntatore al tipo '`inode_t`', in quanto il valore rappresentato da **inode_mnt* deve poter essere modificato dalla funzione. Infatti, quando si vuole innestare il file system principale, si crea una situazione particolare, perché la directory di innesto è la radice dello stesso file system da innestare; pertanto, **inode_mnt* deve essere un puntatore nullo ed è compito della funzione far sì che diventi il puntatore alla voce corretta nella tabella degli inode.

Questa funzione viene usata da *proc_init(9)* [93.20.3] per innestare il file system principale, e da *s_mount(9)* [93.12] per innestare un file system in condizioni diverse.

VALORE RESTITUITO

La funzione restituisce il puntatore all'elemento della tabella dei super blocchi che rappresenta il dispositivo innestato. In caso

si insuccesso, restituisce invece il puntatore nullo e aggiorna la variabile *errno* del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EBUSY	Il dispositivo richiesto risulta già innestato; la directory di innesto è già utilizzata; la tabella dei super blocchi è già occupata del tutto.
EIO	Errore di input-output.
ENODEV	Il file system del dispositivo richiesto non può essere gestito.
E_MAP_INODE_TOO_BIG	La mappa che rappresenta lo stato di utilizzo degli inode del file system, è troppo grande e non può essere caricata in memoria.
E_MAP_ZONE_TOO_BIG	La mappa che rappresenta lo stato di utilizzo delle zone (i blocchi di dati del file system Minix 1) è troppo grande e non può essere caricata in memoria.
E_DATA_ZONE_TOO_BIG	Nel file system che si vorrebbe innestare, la dimensione della zona di dati è troppo grande rispetto alle possibilità di os32.
EUNKNOWN	Errore imprevisto e sconosciuto.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/sb_mount.c’ [94.5.33]

VEDERE ANCHE

sb_reference(9) [93.6.29], *dev_io(9)* [93.4.1], *inode_get(9)* [93.6.15].

93.6.28 os32: sb_print(9)

<<

NOME

‘**sb_print**’ - visualizzazione diagnostica della tabella dei super blocchi

SINTASSI

```
<kernel/fs.h>
void sb_print (void);
```

DESCRIZIONE

La funzione *sb_print()* visualizza sinteticamente i contenuti della tabella dei super blocchi, per fini diagnostici.

FILE SORGENTI

‘kernel/fs.h’ [[94.5](#)]

‘kernel/fs/sb_print.c’ [[94.5.34](#)]

VEDERE ANCHE

inode_print(9) [[93.6.19](#)], *zone_print(9)* [[93.6.36](#)].

93.6.29 os32: sb_reference(9)

<<

NOME

‘**sb_reference**’ - riferimento a un elemento della tabella dei super blocchi

SINTASSI

```
<kernel/fs.h>
sb_t *sb_reference (dev_t device);
```


ARGOMENTI

Argomento	Descrizione
<code>dev_t</code> <i>device</i>	Dispositivo di un'unità di memorizzazione di massa.

DESCRIZIONE

La funzione *sb_reference()* serve a produrre il puntatore a una voce della tabella dei super blocchi. Se si fornisce il numero di un dispositivo già innestato nella tabella, si intende ottenere il puntatore alla voce relativa; se si fornisce il valore zero, si intende semplicemente avere un puntatore alla prima voce (ovvero all'inizio della tabella); se invece si fornisce il valore -1 , si vuole ottenere il riferimento alla prima voce libera.

VALORE RESTITUITO

La funzione restituisce il puntatore all'elemento della tabella dei super blocchi che soddisfa la richiesta. In caso di errore, restituisce invece un puntatore nullo, ma senza dare informazioni aggiuntive con la variabile *errno*, perché il motivo è implicito nel tipo di richiesta.

ERRORI

In caso di errore la variabile *errno* non viene aggiornata. Tuttavia, se l'errore deriva dalla richiesta di un dispositivo di memorizzazione, significa che non è presente nella tabella; se è stata richiesta una voce libera, significa che la tabella dei super blocchi è occupata completamente.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/fs_public.c' [94.5.7]

'kernel/fs/sb_reference.c' [94.5.35]

VEDERE ANCHE

inode_reference(9) [93.6.21], *file_reference(9)* [93.6.5].

93.6.30 os32: *sb_save(9)*

<<

NOME

'**sb_save**' - memorizzazione di un super blocco nel proprio file system

SINTASSI

```
<kernel/fs.h>
int sb_save (sb_t *sb);
```

ARGOMENTI

Argomento	Descrizione
sb_t *sb	Puntatore a una voce della tabella dei super blocchi in memoria.

DESCRIZIONE

La funzione *sb_save()* verifica se il super blocco conservato in memoria e rappresentato dal puntatore *sb* risulta modificato; in tal caso provvede ad aggiornarlo nell'unità di memorizzazione di origine, assieme alle mappe di utilizzo degli inode e delle zone di dati.

VALORE RESTITUITO

Valore	Significato
0	Operazione conclusa con successo.
-1	Errore: la variabile <i>errno</i> viene impostata di conseguenza.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Il riferimento al super blocco è un puntatore nullo.
EIO	Errore di input-output.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/sb_save.c’ [94.5.36]

VEDERE ANCHE

inode_save(9) [93.6.22], *dev_io(9)* [93.4.1].

93.6.31 os32: *sb_zone_status(9)*

Vedere *sb_inode_status(9)* [93.6.26].

93.6.32 os32: *sock_free_port(9)*

NOME

‘**sock_free_port**’ - scansione della tabella dei socket alla ricerca di una porta libera, da 1024 in su

SINTASSI

```
<kernel/fs.h>
h_port_t sock_free_port (void);
```

DESCRIZIONE

La funzione *sock_free_port()* restituisce il numero di una porta libera, dopo avere scandito la tabella dei socket. La ricerca riguarda esclusivamente le porte da 1024 in su, ovvero di quelle

che possono essere utilizzate da processi non privilegiati. Nel caso le porte siano tutte impegnate, la funzione restituisce il valore zero.

Il tipo `'h_port_t'` a cui si riferisce il valore restituito dalla funzione è un valore scalare, adatto a contenere il numero di una porta, rappresentato secondo l'ordinamento dei byte del sistema (*host byte order*).

VALORE RESTITUITO

La funzione restituisce il numero della porta libera trovata, oppure zero in mancanza di questo.

FILE SORGENTI

`'kernel/fs.h'` [94.5]

`'kernel/fs/sock_free_port.c'` [94.5.38]

VEDERE ANCHE

`sock_reference(9)` [93.6.33].

93.6.33 os32: `sock_reference(9)`

«

NOME

`'sock_reference'` - riferimento a un elemento della tabella dei socket

SINTASSI

```
<kernel/fs.h>
sock_t *sock_reference (int skn);
```

ARGOMENTI

Argomento	Descrizione
<code>int <i>skn</i></code>	Indice all'interno della tabella dei socket.

DESCRIZIONE

La funzione *sock_reference()* serve a produrre il puntatore a una voce della tabella dei super blocchi. Se si fornisce un indice positivo (maggiore o uguale a zero), si ottiene *&sock_table[skn]*; altrimenti, con un valore negativo qualunque, viene scandita la tabella alla ricerca della prima voce libera, di cui si restituisce il puntatore allo stesso modo.

VALORE RESTITUITO

La funzione restituisce il puntatore all'elemento della tabella dei super blocchi che soddisfa la richiesta. Non si manifestano errori e se viene richiesto un indice positivo troppo grande, si ottiene un puntatore al di fuori della tabella.

FILE SORGENTI

'kernel/fs.h' [[94.5](#)]

'kernel/fs/fs_public.c' [[94.5.7](#)]

'kernel/fs/sock_reference.c' [[94.5.39](#)]

VEDERE ANCHE

sock_free_port(9) [[93.6.32](#)].

93.6.34 os32: zone_alloc(9)

<<

NOME‘**zone_alloc**’, ‘**zone_free**’ - allocazione di zone di dati**SINTASSI**

```
<kernel/fs.h>
zno_t zone_alloc (sb_t *sb);
int zone_free (sb_t *sb, zno_t zone);
```

ARGOMENTI

Argomento	Descrizione
sb_t *sb	Puntatore a una voce della tabella dei super blocchi in memoria.
zno_t zone	Numero di zona da liberare.

DESCRIZIONE

La funzione *zone_alloc()* occupa una zona nella mappa associata al super blocco a cui si riferisce *sb*, restituendone il numero. La funzione *zone_free()* libera una zona che precedentemente risultava occupata nella mappa relativa.

VALORE RESTITUITO

La funzione *zone_alloc()* restituisce il numero della zona allocata. Se questo numero è zero, si tratta di un errore, e va considerato il contenuto della variabile *errno*.

La funzione *zone_free()* restituisce zero in caso di successo, oppure -1 in caso di errore, aggiornando di conseguenza la variabile *errno*.

ERRORI

Valore di <i>errno</i>	Significato
EROFS	Il file system è innestato in sola lettura, pertanto non è possibile apportare cambiamenti alla mappa di utilizzo delle zone.
ENOSPC	Non è possibile allocare una zona, perché non ce ne sono di libere.
EINVAL	L'argomento corrispondente a <i>sb</i> è un puntatore nullo; la zona di cui si richiede la liberazione è precedente alla prima zona dei dati (pertanto non può essere liberata, in quanto riguarda i dati amministrativi del super blocco); la zona da liberare è successiva allo spazio gestito dal file system.
EUNKNOWN	Errore imprevisto e sconosciuto.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/zone_alloc.c' [94.5.40]

'kernel/fs/zone_free.c' [94.5.41]

VEDERE ANCHE

zone_write(9) [93.6.37], *sb_save(9)* [93.6.30].

93.6.35 os32: *zone_free(9)*

Vedere *zone_alloc(9)* [93.6.34].

93.6.36 os32: zone_print(9)

<<

NOME

‘**zone_print**’ - visualizzazione diagnostica del contenuto della prima parte di una zona dati

SINTASSI

```
<kernel/fs.h>
void zone_print (sb_t *sb, zno_t zone);
```

DESCRIZIONE

La funzione *zone_print()* visualizza la prima parte del contenuto di una zona dati, riferita a un certo super blocco, in esadecimale, per fini diagnostici.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/zone_print.c’ [94.5.42]

VEDERE ANCHE

inode_print(9) [93.6.19], *sb_print(9)* [93.6.28].

93.6.37 os32: zone_read(9)

<<

NOME

‘**zone_read**’, ‘**zone_write**’ - lettura o scrittura di una zona di dati

SINTASSI

```
<kernel/fs.h>
int zone_read (sb_t *sb, zno_t zone, void *buffer);
int zone_write (sb_t *sb, zno_t zone, void *buffer);
```

ARGOMENTI

Argomento	Descrizione
sb_t *sb	Puntatore a una voce della tabella dei super blocchi in memoria.
zno_t zone	Numero di zona da leggere o da scrivere
void *buffer	Puntatore alla posizione iniziale in memoria dove depositare la zona letta o da dove trarre i dati per la scrittura della zona.

DESCRIZIONE

La funzione *zone_read()* legge una zona e ne trascrive il contenuto a partire da *buffer*. La funzione *zone_write()* scrive una zona copiandovi al suo interno quanto si trova in memoria a partire da *buffer*. La zona è individuata dal numero *zone* e riguarda il file system a cui si riferisce il super blocco *sb*.

La lettura o la scrittura riguarda una zona soltanto, ma nella sua interezza.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	Gli argomenti non sono validi.
EROFS	Il file system è innestato in sola lettura.
EIO	Errore di input-output.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/zone_read.c’ [94.5.43]

‘kernel/fs/zone_write.c’ [94.5.44]

VEDERE ANCHE

zone_alloc(9) [93.6.34], *zone_free(9)* [93.6.34].

93.6.38 os32: *path_device(9)*

«

NOME

‘**path_device**’ - conversione di un file di dispositivo nel numero corrispondente

SINTASSI

```
<kernel/fs.h>
dev_t path_device (pid_t pid, const char *path);
```

ARGOMENTI

Argomento	Descrizione
<code>pid_t <i>pid</i></code>	Processo elaborativo per conto del quale si agisce.
<code>const char *<i>path</i></code>	Il percorso del file di dispositivo.

DESCRIZIONE

La funzione `path_device()` consente di trarre il numero complessivo di un dispositivo, a partire da un file di dispositivo.

Questa funzione viene usata soltanto da `s_mount(9)` [93.12].

VALORE RESTITUITO

La funzione restituisce il numero del dispositivo corrispondente al file indicato, oppure il valore `-1`, in caso di errore, aggiornando la variabile `errno` del kernel.

ERRORI

Valore di <code>errno</code>	Significato
<code>ENODEV</code>	Il file richiesto non è un file di dispositivo.
<code>ENOENT</code>	Il file richiesto non esiste.
<code>EACCES</code>	Il file richiesto non è accessibile secondo i privilegi del processo <code>pid</code> .

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/path_device.c’ [94.5.27]

VEDERE ANCHE

proc_reference(9) [93.20.5], *path_inode(9)* [93.6.41],
inode_put(9) [93.6.20].

93.6.39 os32: path_fix(9)

<<

NOME

‘**path_fix**’ - semplificazione di un percorso

SINTASSI

```
<kernel/fs.h>
int path_fix (char *path);
```

ARGOMENTI

Argomento	Descrizione
char * <i>path</i>	Il percorso da semplificare.

DESCRIZIONE

La funzione *path_fix()* legge la stringa del percorso *path* e la rielabora, semplificandolo. La semplificazione riguarda l’eliminazione di riferimenti inutili alla directory corrente e di indietro-giamenti. Il percorso può essere assoluto o relativo: la funzione non ne cambia l’origine.

VALORE RESTITUITO

La funzione restituisce sempre zero e non è prevista la manifestazione di errori.

FILE SORGENTI

‘kernel/fs.h’ [94.5]

‘kernel/fs/path_fix.c’ [94.5.28]

VEDERE ANCHE

strtok(3) [88.129], *strcmp(3)* [88.115], *strcat(3)* [88.113],
strncat(3) [88.113], *strncpy(3)* [88.117].

93.6.40 os32: path_full(9)



NOME

‘**path_full**’ - traduzione di un percorso relativo in un percorso assoluto

SINTASSI

```
<kernel/fs.h>
int path_full (const char *path, const char *path_cwd,
               char *full_path);
```

ARGOMENTI

Argomento	Descrizione
const char * <i>path</i>	Il percorso relativo alla posizione <i>path_cwd</i> .
const char * <i>path_cwd</i>	La directory corrente.
char * <i>full_path</i>	Il luogo in cui scrivere il percorso assoluto.

DESCRIZIONE

La funzione *path_full()* ricostruisce un percorso assoluto, mettendolo in memoria a partire da ciò a cui punta *full_path*.

VALORE RESTITUITO

Valore	Significato del valore restituito
0	Operazione conclusa con successo.
-1	Errore, descritto dal contenuto della variabile <i>errno</i> del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	L'insieme degli argomenti non è valido.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/path_full.c' [94.5.29]

VEDERE ANCHE

strtok(3) [88.129], *strcmp(3)* [88.115], *strcat(3)* [88.113], *strncat(3)* [88.113], *strncpy(3)* [88.117], *path_fix(9)* [93.6.39].

93.6.41 os32: path_inode(9)

«

NOME

'**path_inode**' - caricamento di un inode, partendo dal percorso del file

SINTASSI

```
<kernel/fs.h>
inode_t *path_inode (pid_t pid, const char *path);
```

ARGOMENTI

Argomento	Descrizione
<code>pid_t <i>pid</i></code>	Il numero del processo elaborativo per conto del quale si agisce.
<code>const char *<i>path</i></code>	Il percorso del file del quale si vuole ottenere l'inode.

DESCRIZIONE

La funzione `path_inode()` carica un inode nella tabella degli inode, oppure lo localizza se questo è già caricato, partendo dal percorso di un file. L'operazione è subordinata all'accessibilità del percorso che conduce al file, nel senso che il processo `pid` deve avere il permesso di accesso («x») in tutti gli stadi dello stesso.

VALORE RESTITUITO

La funzione restituisce il puntatore all'elemento della tabella degli inode che contiene le informazioni caricate in memoria sull'inode. Se qualcosa non va, restituisce invece il puntatore nullo, aggiornando di conseguenza il contenuto della variabile `errno` del kernel.

ERRORI

Valore di <code>errno</code>	Significato
ENOENT	Uno dei componenti del percorso non esiste.
ENFILE	Non è possibile allocare un inode ulteriore, perché la tabella è già occupata completamente.
EIO	Error di input-output.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/path_inode.c' [94.5.30]

VEDERE ANCHE

proc_reference(9) [93.20.5], *path_full(9)* [93.6.40], *inode_get(9)* [93.6.15], *inode_put(9)* [93.6.20], *inode_check(9)* [93.6.8], *inode_file_read(9)* [93.6.10].

93.6.42 os32: path_inode_link(9)

<<

NOME

'**path_inode_link**' - creazione di un collegamento fisico o di un nuovo file

SINTASSI

```
<kernel/fs.h>
inode_t *path_inode_link (pid_t pid, const char *path,
                          inode_t *inode, mode_t mode);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Il numero del processo elaborativo per conto del quale si agisce.
const char * <i>path</i>	Il percorso del file per il quale si vuole creare il collegamento fisico.
inode_t * <i>inode</i>	Puntatore a una voce della tabella degli inode, alla quale si vuole collegare il nuovo file.
mode_t <i>mode</i>	Nel caso l'inode non sia stato fornito, dovendo creare un nuovo file, questo parametro richiede il tipo e i permessi del file da creare.

DESCRIZIONE

La funzione *path_inode_link()* crea un collegamento fisico con il nome fornito in *path*, riferito all'inode a cui punta *inode*. Tuttavia, l'argomento corrispondente al parametro *inode* può essere un puntatore nullo, e in tal caso viene creato un file vuoto, allocando contestualmente un nuovo inode, usando l'argomento corrispondente al parametro *mode* per il tipo e la modalità dei permessi del nuovo file.

Il processo *pid* deve avere i permessi di accesso per tutte le directory che portano al file da collegare o da creare; inoltre, nell'ultima directory ci deve essere anche il permesso di scrittura, dovendo intervenire sulla stessa modificandola.

VALORE RESTITUITO

La funzione restituisce il puntatore all'elemento della tabella degli inode che descrive l'inode collegato o creato. In caso di problemi, restituisce invece il puntatore nullo, aggiornando di conseguenza il contenuto della variabile *errno* del kernel.

ERRORI

Valore di <i>errno</i>	Significato
EINVAL	L'insieme degli argomenti non è valido: se l'inode è stato indicato, il parametro <i>mode</i> deve essere nullo; al contrario, se l'inode non è specificato, il parametro <i>mode</i> deve contenere informazioni valide.
EPERM	Non è possibile creare il collegamento di un inode corrispondente a una directory.
EMLINK	Non è possibile creare altri collegamenti all'inode, il quale ha già raggiunto la quantità massima.
EEXIST	Il file <i>path</i> esiste già.
EACCES	Impossibile accedere al percorso che dovrebbe contenere il file da collegare.
EROFS	Il file system è innestato in sola lettura e non si può creare il collegamento.

FILE SORGENTI

'kernel/fs.h' [94.5]

'kernel/fs/path_inode_link.c' [94.5.31]

VEDERE ANCHE

proc_reference(9) [93.20.5], *path_inode(9)* [93.6.41], *inode_get(9)* [93.6.15], *inode_put(9)* [93.6.20], *inode_save(9)* [93.6.22], *inode_check(9)* [93.6.8], *inode_alloc(9)* [93.6.7], *inode_file_read(9)* [93.6.10], *inode_file_write(9)* [93.6.11].

93.7 os32: ibm_i386(9)

Il file ‘kernel/ibm_i386.h’ [94.6] descrive le funzioni e le macroistruzioni per la gestione dell’hardware e delle interruzioni, secondo l’architettura IBM i386.

La sezione 84.3 descrive complessivamente queste funzioni e le tabelle successive sono tratte da lì.

Tabella 84.28. Funzioni e macroistruzioni per l’input e l’output con le porte interne x86.

Funzione o macroistruzione	Descrizione
<pre>uint32_t _in_8 (uint32_t <i>port</i>); unsigned int in_8 (<i>port</i>);</pre>	<p>Legge un valore a 8 bit da una porta. Listati 94.6 e 94.6.3.</p>
<pre>uint32_t _in_16 (uint32_t <i>port</i>); unsigned int in_16 (<i>port</i>);</pre>	<p>Legge un valore a 16 bit da una porta. Listati 94.6 e 94.6.1.</p>
<pre>void _out_8 (uint32_t <i>port</i>, uint32_t <i>value</i>); void out_8 (<i>port</i>, <i>value</i>);</pre>	<p>Scrive un valore a 8 bit in una porta. Listati 94.6 e 94.6.6.</p>
<pre>void _out_16 (uint32_t <i>port</i>, uint32_t <i>value</i>); void out_16 (<i>port</i>, <i>value</i>);</pre>	<p>Scrive un valore a 16 bit in una porta. Listati 94.6 e 94.6.4.</p>

Tabella 84.29. Funzioni accessorie per il controllo delle interruzioni hardware.

Funzione o macroistruzione	Descrizione
<code>void cli (void);</code>	Disabilita le interruzioni hardware attraverso l'azzeramento dell'indicatore relativo nel registro EFLAGS. Listati 94.6 e 94.6.7 .
<code>void sti (void);</code>	Abilita le interruzioni hardware attraverso l'attivazione dell'indicatore relativo nel registro EFLAGS. Listati 94.6 e 94.6.27 .
<code>void irq_on (unsigned int <i>irq</i>);</code>	Abilita selettivamente l'interruzione hardware indicata per numero (da zero a 16). Listati 94.6 e 94.6.20 .
<code>void irq_off (unsigned int <i>irq</i>);</code>	Disabilita selettivamente l'interruzione hardware indicata per numero (da zero a 16). Listati 94.6 e 94.6.19 .

Tabella 84.32. Funzioni per la gestione della tabella GDT.

Funzione	Descrizione
<pre>void gdt_segment (int <i>segment</i>, uint32_t <i>base</i>, uint32_t <i>limit</i>, bool <i>present</i>, bool <i>code</i>, unsigned char <i>dpl</i>);</pre>	<p>Scrive una voce della tabella GDT, sezionando e ricomponendo i dati come richiesto dal microprocessore.</p> <p>Listato 94.6.12.</p>
<pre>void gdt (void);</pre>	<p>Predisporre e attiva la tabella GDT; per questo si avvale in modo particolare delle funzioni <i>gdt_segment()</i> e di <i>gdt_load()</i>.</p> <p>Listato 94.6.8.</p>
<pre>void gdt_load (void *<i>gdtr</i>);</pre>	<p>Fa sì che il microprocessore carichi la tabella GDT, a partire dal puntatore al registro GDTR; registro che contiene l'informazione della collocazione in memoria della tabella GDT e della sua estensione.</p> <p>Listato 94.6.9.</p>
<pre>void gdt_print (void *<i>gdtr</i>, unsigned int <i>first</i>, unsigned int <i>last</i>);</pre>	<p>Funzione diagnostica, usata per visualizzare il contenuto della tabella GDT in binario.</p> <p>Listato 94.6.10.</p>

Tabella 84.35. Funzioni per la gestione della tabella IDT.

Funzione	Descrizione
<pre>void idt_descriptor (int <i>desc</i> , void *<i>isr</i> , uint16_t <i>selector</i> , bool <i>present</i> , char <i>type</i> , char <i>dpl</i>) ;</pre>	<p>Scrive una voce della tabella IDT, sezionando e ricomponendo i dati come richiesto dal microprocessore.</p> <p>Listato 94.6.14.</p>
<pre>void idt (void) ;</pre>	<p>Predisporre e attiva la tabella IDT; per questo si avvale in modo particolare delle funzioni <i>idt_descriptor()</i> e di <i>idt_load()</i>.</p> <p>Listato 94.6.13.</p>
<pre>void idt_load (void *<i>idtr</i>) ;</pre>	<p>Fa sì che il microprocessore carichi la tabella IDT, a partire dal puntatore al registro IDTR; registro che contiene l'informazione della collocazione in memoria della tabella IDT e della sua estensione.</p> <p>Listato 94.6.16.</p>

Funzione	Descrizione
<pre>void idt_irq_remap (unsigned int <i>offset_1</i>, unsigned int <i>offset_2</i>);</pre>	<p>Modifica la mappatura delle interruzioni hardware (IRQ) spostando il primo gruppo a partire dal valore di <i>offset_1</i> e il secondo gruppo a partire da <i>offset_2</i>. Listato 94.6.15.</p>
<pre>void idt_print (void *<i>idtr</i>, unsigned int <i>first</i> unsigned int <i>last</i>);</pre>	<p>Funzione diagnostica, usata per visualizzare il contenuto della tabella IDT in binario. Listato 94.6.17.</p>

Tabella 84.37. Funzioni per la gestione delle interruzioni.

Funzione	Descrizione
<pre>void isr_<i>n</i> (void);</pre>	<p>Si tratta di procedure attivate dalle interruzioni, dove per esempio <i>isr_33()</i> viene eseguita a seguito del verificarsi dell'interruzione numero 33, la quale ha origine da IRQ 1, ovvero dalla tastiera. L'indicazione di quale procedura attivare per ogni interruzione dipende dalla configurazione della tabella IDT. Listato 94.6.21.</p>

Funzione	Descrizione
<pre> void isr_exception_unrecoverable (uint32_t <i>eax</i>, uint32_t <i>ecx</i>, uint32_t <i>edx</i>, uint32_t <i>ebx</i>, uint32_t <i>ebp</i>, uint32_t <i>esi</i>, uint32_t <i>edi</i>, uint32_t <i>ds</i>, uint32_t <i>es</i>, uint32_t <i>fs</i>, uint32_t <i>gs</i>, uint32_t <i>interrupt</i>, uint32_t <i>error</i>, uint32_t <i>eip</i>, uint32_t <i>cs</i>, uint32_t <i>eflags</i>); </pre>	<p>Questa funzione viene usata all'interno del file 'isr.s' per segnalare il verificarsi di un'eccezione non risolvibile, come nel caso di una divisione per zero. Pertanto, la funzione ha soprattutto un significato diagnostico. Listato 94.6.23.</p>
<pre> char *isr_exception_name (int <i>exception</i>); </pre>	<p>Restituisce il puntatore alla stringa contenente il nome dell'eccezione corrispondente al numero di interruzione fornito. Viene usata da <i>isr_exception_unrecoverable()</i> per dare delle indicazioni comprensibili sull'eccezione che si è verificata. Listato 94.6.22.</p>

Funzione	Descrizione
<pre>void isr_irq_clear (uint32_t <i>idtn</i>);</pre>	<p>Avvisa il PIC (<i>programmable interrupt controller</i>) che l'interruzione hardware emessa è stata recepita e se ne possono ricevere altre. Tuttavia, essendoci due PIC, la funzione stabilisce quale dei due è coinvolto direttamente e di conseguenza come procedere. Listato 94.6.24.</p>
<pre>void isr_irq_clear_pic1 (void);</pre>	<p>Avvisa il PIC1 che l'interruzione hardware emessa è stata recepita e se ne possono ricevere altre. Listato 94.6.25.</p>
<pre>void isr_irq_clear_pic2 (void);</pre>	<p>Avvisa il PIC2 che l'interruzione hardware emessa è stata recepita e se ne possono ricevere altre. Listato 94.6.26.</p>

93.8 os32: icmp(9)

Il file 'kernel/net/icmp.h' [\[94.12.10\]](#) descrive le funzioni per la gestione del protocollo ICMP. «

Per la descrizione sulla gestione del protocollo ICMP da parte di os32, si rimanda alla sezione [84.11](#). La tabella successiva che sintetizza l'uso delle funzioni di questo gruppo, è tratta da lì.

93.9 os32: ip(9)

«

Il file ‘kernel/net/ip.h’ [94.12.15] descrive le funzioni per la gestione del protocollo IPv4, esclusa però la questione degli instradamenti.

Per la descrizione sulla gestione del protocollo IPv4 eseguita da os32, si rimanda alla sezione 84.10. La tabella successiva che sintetizza l’uso delle funzioni di questo gruppo, è tratta da lì.

Tabella 84.124. Funzioni per la gestione dei pacchetti a livello IP.

Funzione	Descrizione
<pre>uint16_t ip_checksum (uint16_t *data1, size_t size1, uint16_t *data2, size_t size2);</pre>	<p>Produce il codice di controllo usato nel protocollo IPv4, partendo da due blocchi di dati [94.12.16].</p>
<pre>int ip_rx (int n, int f);</pre>	<p>Si occupa di acquisire un pacchetto IPv4, dalla tabella <i>net_table[]</i>, per copiarlo nella tabella <i>ip_table[]</i> e trattarlo per le questioni urgenti [94.12.21].</p>
<pre>ip_t *ip_reference (void);</pre>	<p>Restituisce il puntatore a un elemento della tabella <i>ip_table[]</i> che possa essere riutilizzato, perché mai usato prima oppure perché contenente il pacchetto IPv4 ricevuto che risulta essere più vecchio di tutti gli altri [94.12.20].</p>

Funzione	Descrizione
<pre>ssize_t ip_header (h_addr_t <i>src</i>, h_addr_t <i>dst</i>, uint16_t <i>id</i>, uint8_t <i>ttl</i>, uint8_t <i>protocol</i>, void *<i>buffer</i>, size_t <i>length</i>);</pre>	<p>Scrive, in corrispondenza di <i>buffer</i>, un'intestazione IPv4, sulla base dei dati contenuti negli altri parametri [94.12.17].</p>
<pre>int ip_tx (h_addr_t <i>src</i>, h_addr_t <i>dst</i>, int <i>protocol</i>, const void *<i>buffer</i>, size_t <i>size</i>);</pre>	<p>Produce e trasmette un pacchetto IPv4, partendo dal contenuto che deve avere e dai dati necessari a costruire l'intestazione IPv4 [94.12.22].</p>
<pre>h_addr_t ip_mask (int <i>m</i>);</pre>	<p>A partire da un numero che rappresenta la dimensione di una maschera di rete, si ottiene il valore a 32 bit della maschera stessa. Per esempio, dal valore 16, si ottiene 255.255.0.0 [94.12.18].</p>

93.10 os32: kbd(9)

Il file 'kernel/driver/kbd.h' [94.4.15] descrive le funzioni per la gestione della tastiera, in relazione alla gestione complessiva dei terminali virtuali.

Per la descrizione dell'organizzazione della gestione della tastiera di os32, si rimanda alla sezione 84.7.5.1. La tabella successiva che

sintetizza l'uso delle funzioni di questo gruppo, è tratta da lì.

Tabella 84.87. Funzioni per la gestione della tastiera, dichiarate nel file di intestazione 'kernel/driver/kbd.h' e descritte nei file contenuti nella directory 'kernel/driver/kbd/'.

Funzione	Descrizione
<pre>void kbd_isr (void);</pre>	<p>Questa funzione è chiamata dalla routine di gestione delle interruzioni da tastiera, contenuta nel file 'kernel/ibm_i386/isr.s'. La funzione legge un carattere dalla porta di I/O 60₁₆, quindi lo interpreta e aggiorna il contenuto della variabile strutturata <i>kbd</i> di conseguenza.</p>
<pre>void kbd_load (void);</pre>	<p>Questa funzione è chiamata una sola volta da <i>kmain()</i>, per associare la mappa della tastiera ai codici prodotti dalla stessa. Attualmente questa funzione produce esclusivamente l'associazione necessaria per una tastiera italiana.</p>

93.11 os32: lib_k(9)

«

Il file 'kernel/lib_k.h' [94.7] descrive alcune funzioni con nomi che iniziano per 'k_...' (dove la lettera «k» sta per kernel) e riproducono il comportamento di funzioni standard, della libreria C. Per esempio, *k_printf()* è l'equivalente di *printf()*, ma per la gestione interna del kernel.

Teoricamente, quando una funzione interna al kernel può ricondursi allo standard, dovrebbe avere il nome previsto. Tuttavia, per evitare di dover qualificare ogni volta l'ambito di una funzione, sono stati usati nomi differenti, ciò anche al fine di non creare complicazioni in fase di compilazione di tutto il sistema.

93.12 os32: lib_s(9)

Il file `'kernel/lib_s.h'` [94.8] descrive alcune funzioni con nomi che iniziano per `'s_...'` (dove la lettera «s» sta per sistema) e servono a realizzare le chiamate di sistema, all'interno del kernel. Per esempio, la funzione `s__exit()` realizza la funzione `_exit()`. Il prototipo delle funzioni `s_...()` è lo stesso di quelle a cui si riferiscono, con l'aggiunta iniziale del numero del processo da cui ha origine la chiamata di sistema.

VEDERE ANCHE

sysroutine(9) [93.20.28], *brk(2)* [87.5], *chdir(2)* [87.6], *chmod(2)* [87.7], *chown(2)* [87.8], *clock(2)* [87.9], *close(2)* [87.10], *dup2(2)* [87.12], *dup(2)* [87.12], *_exit(2)* [87.2], *fchmod(2)* [87.7], *fchown(2)* [87.8], *fcntl(2)* [87.18], *fork(2)* [87.19], *fstat(2)* [87.55], *kill(2)* [87.29], *link(2)* [87.30], *longjmp(2)* [87.49], *lseek(2)* [87.33], *mkdir(2)* [87.34], *mknod(2)* [87.35], *mount(2)* [87.36], *open(2)* [87.37], *pipe(2)* [87.38], *read(2)* [87.39], *sbrk(2)* [87.5], *setegid(2)* [87.48], *seteuid(2)* [87.51], *setgid(2)* [87.48], *setjmp(2)* [87.49], *setuid(2)* [87.51], *signal(2)* [87.52], *stat(2)* [87.55], *stime(2)* [87.59], *tcgetattr(2)* [87.58], *tcsetattr(2)*

[87.58], *time(2)* [87.59], *umount(2)* [87.36], *unlink(2)* [87.62], *wait(2)* [87.63], *write(2)* [87.64].

93.13 os32: main(9)

«

Il file ‘kernel/main.h’ [94.9] descrive la funzione *kmain()* del kernel e altre funzioni accessorie, assieme al codice iniziale necessario per mettere in funzione il kernel stesso.

Si rimanda alla sezione 84.2 che descrive dettagliatamente il codice iniziale del kernel.

93.14 os32: memory(9)

«

Il file ‘kernel/memory.h’ [94.10] descrive le funzioni per la gestione della memoria, a livello di sistema.

Per la descrizione dell’organizzazione della gestione della memoria si rimanda alla sezione 84.6. Le tabelle successive che sintetizzano l’uso delle funzioni di questo gruppo, sono tratte da quel capitolo.

Tabella 84.77. Funzioni per la gestione della mappa della memoria, dichiarate nel file di intestazione ‘kernel/memory.h’ e realizzate nella directory ‘kernel/memory/’.

Funzione	Descrizione
<pre>uint32_t *mb_reference (void);</pre>	Restituisce il puntatore alla tabella dei blocchi di memoria, per uniformare l'accesso alla tabella dalle funzioni che non fanno parte del gruppo contenuto nella directory 'kernel/memory/'.
<pre>ssize_t mb_alloc (addr_t <i>address</i>, size_t <i>size</i>);</pre>	Alloca la memoria a partire dall'indirizzo indicato, per la quantità di byte richiesta. L'allocazione ha termine anticipatamente se si incontra un blocco già utilizzato. La funzione restituisce la dimensione allocata effettivamente.
<pre>void mb_free (addr_t <i>address</i>, size_t <i>size</i>);</pre>	Libera la memoria a partire dall'indirizzo indicato, per la quantità di byte richiesta. Lo spazio viene liberato in ogni caso, anche se risulta già libero; tuttavia viene prodotto un avvertimento a video se si verifica tale ipotesi.

Funzione	Descrizione
<pre>int mb_reduce (addr_t <i>address</i>, size_t <i>new</i>, size_t <i>previous</i>);</pre>	Riduce un'area di memoria già utilizzata. Restituisce zero se l'operazione si conclude con successo, oppure -1 in caso contrario, aggiornando la variabile <i>errno</i> di conseguenza.
<pre>void mb_clean (addr_t <i>address</i>, size_t <i>size</i>);</pre>	Azzera l'area di memoria specificata.
<pre>addr_t mb_alloc_size (size_t <i>size</i>);</pre>	Alloca un'area di memoria della dimensione richiesta, restituendone l'indirizzo. La funzione conclude con successo il proprio lavoro se il valore restituito è diverso da zero; se invece l'indirizzo ottenuto è pari a zero si è verificato un errore che può essere verificato analizzando il contenuto della variabile <i>errno</i> .

Funzione	Descrizione
<pre>void mb_size (size_t <i>size</i>);</pre>	<p>Questa funzione, usata una sola volta all'interno di <i>kmain()</i>, serve a definire la dimensione massima della memoria disponibile in blocchi. In pratica, le si fornisce la dimensione effettiva della memoria che viene così divisa per la dimensione del blocco, ignorando il resto. Questa informazione viene conservata nella variabile <i>mb_max</i>.</p>
<pre>void mb_print (void);</pre>	<p>Funzione diagnostica che visualizza gli intervalli di memoria utilizzati, esprimendoli però in blocchi.</p>

93.15 os32: multiboot(9)

Il file 'kernel/multiboot.h' [94.11] descrive delle funzioni e un tipo derivato per il trattamento delle informazioni multiboot. «

Per la descrizione della gestione delle informazioni multiboot da parte di os32, si rimanda alla sezione 84.2.1; la tabella successiva descrive brevemente le due funzioni disponibili per questa gestione.

Tabella 84.14. Funzioni per la gestione delle specifiche multiboot all'interno di os32.

Funzione	Descrizione
<pre>void mboot_save (multiboot_t *<i>mboot_data</i>);</pre>	<p>Salva le informazioni multiboot all'interno della variabile strutturata pubblica <i>multiboot</i>. Listati 94.11, 94.11.2 e 94.11.3.</p>
<pre>char ** mboot_cmdline_opt (const char *<i>opt</i>, const char *<i>delim</i>);</pre>	<p>Scandisce la stringa delle opzioni salvata all'interno di <i>multiboot.cmdline</i>, alla ricerca di un'opzione il cui nome corrisponda alla stringa. Dopo il nome dell'opzione deve apparire il segno '=' e dopo devono trovarsi i valori associati all'opzione, separati da <i>delim</i>. Questi valori vengono restituiti in forma di array di stringhe, dove l'ultima stringa si riconosce perché vuota. Listati 94.11, 94.11.2 e 94.11.1.</p>

93.16 os32: ne2k(9)

Il file ‘kernel/driver/nic/ne2k.h’ [94.4.19] descrive le funzioni per la gestione delle interfacce di rete NE2K. «

Per la descrizione della gestione dei dispositivi NE2K si rimanda alla sezione 84.9.1. La tabella successiva che sintetizza l’uso delle funzioni di questo gruppo, sono tratte da quella sezione.

Tabella 84.118. Funzioni per la gestione dell’interfaccia di rete NE2000.

Funzione	Descrizione
<code>int ne2k_check (uintptr_t io);</code>	Verifica se l’interfaccia corrispondente alla porta di I/O specificata è veramente di tipo NE2000 [94.4.20].
<code>int ne2k_isr (uintptr_t io);</code>	Verifica lo stato dell’interfaccia e ne acquisisce i dati, se disponibili. In caso di ricezione di una trama, viene chiamata la funzione <i>ne2k_rx()</i> per trasferirla nella memoria tampone della tabella delle interfacce [94.4.21].

Funzione	Descrizione
<pre>int ne2k_isr_expect (uintptr_t <i>io</i>, unsigned int <i>isr_expect</i>);</pre>	<p>Rimane in attesa fino a che il registro ISR dell'interfaccia si attiva almeno un indicatore corrispondente a quanto richiesto con il parametro <i>isr_expect</i>. Questa funzione viene usata, in particolare, nella trasmissione dei pacchetti, per i quali occorre verificare quando l'interfaccia ha completato il procedimento [94.4.22].</p>
<pre>int ne2k_reset (uintptr_t <i>io</i>, void *<i>address</i>);</pre>	<p>Azzerare l'interfaccia e ne estrae l'indirizzo fisico, collocandolo in corrispondenza di <i>*address</i> [94.4.23].</p>
<pre>int ne2k_rx (uintptr_t <i>io</i>);</pre>	<p>Copia tutte le trame accumulate nella memoria tampone interna dell'interfaccia, in quella della tabella delle interfacce [94.4.24].</p>
<pre>int ne2k_rx_reset (uintptr_t <i>io</i>);</pre>	<p>Reinizializza il processo di ricezione [94.4.25].</p>

Funzione	Descrizione
<pre data-bbox="108 369 946 482">int ne2k_tx (uintptr_t <i>io</i>, void *<i>buffer</i>, size_t <i>size</i>);</pre>	<p data-bbox="970 157 1490 662">Trasmette una trama Ethernet, contenuta all'interno di <i>*buffer</i>, della lunghezza specificata da <i>size</i>. La funzione attende il completamento dell'operazione, prima di concludere il proprio funzionamento [94.4.26].</p>

93.17 os32: net(9)

Il file 'kernel/net.h' [94.12] descrive le funzioni per la gestione della tabella delle interfacce, assieme ad altre funzioni accessorie relative alla trasmissione di trame Ethernet. «

Per la descrizione sulla gestione della tabella delle interfacce, si rimanda alla sezione 84.9.2. La tabella successiva che sintetizza l'uso delle funzioni di questo gruppo, è tratta da lì.

Tabella 84.121. Funzioni per la gestione della tabella delle interfacce, contenute nella directory 'kernel/net/', e altre accessorie relative alla gestione Ethernet.

Funzione	Descrizione
<pre>net_buffer_eth_t * net_buffer_eth (int <i>n</i>);</pre>	<p>Restituisce il puntatore a un elemento libero, o utilizzabile, della memoria tampone dell'interfaccia 'net<i>n</i>', purché questa sia di tipo Ethernet [94.12.23].</p>
<pre>net_buffer_lo_t * net_buffer_lo (int <i>n</i>);</pre>	<p>Restituisce il puntatore a un elemento libero, o utilizzabile, della memoria tampone dell'interfaccia 'net<i>n</i>', purché questa sia di tipo <i>loopback</i>, ossia l'interfaccia locale virtuale [94.12.24].</p>
<pre>int net_index (h_addr_t <i>ip</i>);</pre>	<p>Restituisce l'indice della tabella delle interfacce, corrispondente all'indirizzo IPv4 fornito come argomento [94.12.27].</p>
<pre>int net_index_eth (h_addr_t <i>ip</i>, uint8_t <i>mac</i>[6], uintptr_t <i>io</i>);</pre>	<p>Restituisce l'indice della tabella delle interfacce, corrispondente a uno dei dati forniti come argomento (i valori nulli vengono ignorati), purché si tratti di un'interfaccia Ethernet [94.12.28].</p>

Funzione	Descrizione
<pre>void net_init (void);</pre>	<p>Inizializza la gestione della rete, utilizzando le informazioni attraverso le opzioni di avvio per configurare anche le interfacce Ethernet [94.12.29].</p>
<pre>void net_rx (void);</pre>	<p>Scandisce i pacchetti memorizzati nella tabella delle interfacce, passandoli al gestore appropriato e rimuovendoli poi dalla tabella [94.12.32].</p>
<pre>int net_eth_ip_tx (h_addr_t <i>src</i>, h_addr_t <i>dst</i>, const void *<i>packet</i>, size_t <i>size</i>);</pre>	<p>A partire da un pacchetto IPv4 completo e dagli indirizzi IPv4 di origine e di destinazione, viene assemblata e spedita una trama Ethernet. La funzione richiede separatamente l'indicazione degli indirizzi IPv4 di origine e destinazione, per semplificare il codice, evitando di estrapolarli dal pacchetto IPv4 stesso [94.12.25].</p>

Funzione	Descrizione
<pre>int net_eth_tx (int <i>n</i>, void *<i>buffer</i>, size_t <i>size</i>);</pre>	Provvede a trasmettere una trama Ethernet attraverso l'interfaccia <i>n</i> (ovvero <i>net_table[n]</i>), la quale deve essere di tipo Ethernet [94.12.26].

93.18 os32: part(9)

«

Il file ‘kernel/part.h’ [94.13] descrive la struttura ‘**part_t**’ e delle macro-variabili per la gestione delle partizioni. Solo la funzione *dm_init()* si avvale di questo file di intestazione.

93.19 os32: pci(9)

«

Il file ‘kernel/driver/pci.h’ [94.4.27] descrive le funzioni per la gestione del bus PCI; ma in pratica è disponibile solo *pci_init()* che ha lo scopo di scandire il bus per trovare i dispositivi presenti e annotarli nella tabella PCI, funzione che viene usata una volta sola, all'avvio del sistema. A tale proposito va osservato che os32 è in grado di leggere solo il bus principale e non può aggiornare la tabella dei dispositivi.

Per una descrizione ulteriore del funzionamento del bus PCI, nell'ottica semplificata di os32, si veda la sezione 83.10.

93.20 os32: proc(9)

Il file ‘kernel/proc.h’ [94.14] descrive ciò che serve per la gestione dei processi. In modo particolare, in questo file si definisce il tipo derivato ‘**proc_t**’, con cui si realizza la tabella dei processi.

Si veda in particolare la spiegazione contenuta nella sezione 84.4 al riguardo della gestione dei processi.

93.20.1 os32: proc_available(9)

NOME

‘**proc_available**’ - inizializzazione di un processo libero

SINTASSI

```
<kernel/proc.h>
void proc_available (pid_t pid);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Il numero del processo elaborativo da inizializzare.

DESCRIZIONE

La funzione *proc_available()* si limita a inizializzare, con valori appropriati, i dati di un processo nella tabella relativa, in modo che risulti correttamente uno spazio libero per le allocazioni successive.

Questa funzione viene usata da *proc_init(9)* [93.20.3], *proc_sig_chld(9)* [93.20.11], *s_wait(9)* [93.12].

FILE SORGENTI

‘kernel/proc.h’ [94.14]

‘kernel/proc/proc_public.c’ [94.14.5]

‘kernel/proc/proc_available.c’ [94.14.1]

93.20.2 os32: proc_dump_memory(9)

«

NOME

‘**proc_dump_memory**’ - copia di una porzione di memoria in un file

SINTASSI

```
<kernel/proc.h>
void proc_dump_memory (pid_t pid, addr_t address, size_t size,
                       char *name);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Il numero del processo elaborativo per conto del quale si agisce.
addr_t <i>address</i>	Indirizzo efficace della memoria.
size_t <i>size</i>	Quantità di byte da trascrivere, a partire dall’indirizzo efficace.
char * <i>name</i>	Nome del file da creare.

DESCRIZIONE

La funzione *proc_dump_memory()* salva in un file una porzione di memoria, secondo le coordinate fornita dagli argomenti.

Viene usata esclusivamente da *proc_sig_core(9)* [93.20.3], quando si riceve un segnale per cui è necessario scaricare la memoria di un processo. In quel caso, se il processo eliminato ha i permessi per scrivere nella directory radice, vengono creati due file: uno con l'immagine del segmento codice ('/core.i') e l'altro con l'immagine del segmento dati ('/core.d').

FILE SORGENTI

'kernel/proc.h' [94.14]

'kernel/proc/proc_sig_core.c' [94.14.14]

93.20.3 os32: proc_init(9)



NOME

'**proc_init**' - inizializzazione della gestione complessiva dei processi elaborativi

SINTASSI

```
<kernel/proc.h>
extern uint32_t  _k_start;
extern uint32_t  _k_end;
extern uint32_t  _k_text_end;
extern uint32_t  _k_data_end;
extern uint32_t  _k_bss_end;
extern uint32_t  _k_stack_top;
extern uint32_t  _k_stack_bottom;
void proc_init (void);
```

ARGOMENTI

Argomento	Descrizione
<code>extern uint32_t <i>_k_start</i>;</code>	La variabile <i>_k_start</i> viene fornita dal file di configurazione ‘kernel.ld’ e rappresenta l’indirizzo iniziale del kernel.
<code>extern uint32_t <i>_k_end</i>;</code>	La variabile <i>_k_end</i> viene fornita dal file di configurazione ‘kernel.ld’ e rappresenta l’indirizzo finale del kernel, dati e pila inclusi.
<code>extern uint32_t <i>_k_text_end</i>;</code>	La variabile <i>_k_text_end</i> viene fornita dal file di configurazione ‘kernel.ld’ e rappresenta l’indirizzo finale del codice del kernel.
<code>extern uint32_t <i>_k_data_end</i>;</code>	La variabile <i>_k_data_end</i> viene fornita dal file di configurazione ‘kernel.ld’ e rappresenta l’indirizzo finale dei dati kernel, esclusa però l’area BSS e la pila.

Argomento	Descrizione
<code>extern uint32_t <i>_k_bss_end</i>;</code>	La variabile <i>_k_bss_end</i> viene fornita dal file di configurazione 'kernel.ld' e rappresenta l'indirizzo finale dell'area BSS del kernel, ma coincide con <i>_k_end</i> .
<code>extern uint32_t <i>_k_stack_top</i>;</code>	La variabile <i>_k_stack_top</i> viene fornita dal file 'kernel/main/stack.s' e rappresenta l'indirizzo iniziale della pila dei dati del kernel.
<code>extern uint32_t <i>_k_stack_bottom</i>;</code>	La variabile <i>_k_stack_bottom</i> viene fornita dal file 'kernel/main/stack.s' e rappresenta l'indirizzo finale della pila dei dati del kernel.

DESCRIZIONE

La funzione *proc_init()* viene usata una volta sola, dalla funzione *kmain(9)* [93.13], per predisporre la gestione dei processi. Per la precisione si occupa di:

- predisporre la tabella GDT, attraverso la chiamata della funzione *gdt()*;
- impostare il temporizzatore in modo da fornire impulsi alla frequenza dichiarata nella macro-variabile *CLOCKS_PER_SEC*, pari a 100 Hz;
- predisporre la tabella IDT, attraverso la chiamata della funzione *idt()*;
- azzerare la tabella dei processi, inserendovi però i dati relativi al kernel (il processo zero);

- allocare la memoria già utilizzata dal kernel;
- attivare selettivamente le interruzioni hardware desiderate;
- attivare la gestione delle unità PATA;
- innestare il file system principale.

FILE SORGENTI

'kernel/proc.h' [94.14]

'kernel/proc/proc_public.c' [94.14.5]

'kernel/proc/proc_init.c' [94.14.3]

VEDERE ANCHE

proc_available(9) [93.20.1], *sb_mount(9)* [93.6.27].

93.20.4 os32: proc_print(9)

«

NOME

'**proc_print**' - visualizzazione diagnostica dei in corso

SINTASSI

```
<kernel/proc.h>  
void proc_print (void);
```

DESCRIZIONE

La funzione *proc_print()* visualizza sinteticamente i processi in corso, per fini diagnostici. Viene usata nella funzione *kmain()*, quando il kernel è in modalità di funzionamento interattivo.

FILE SORGENTI

'kernel/proc.h' [94.14]

'kernel/proc/proc_print.c' [94.14.4]

93.20.5 os32: `proc_reference(9)`**NOME**

‘**proc_reference**’ - puntatore alla voce che rappresenta un certo processo

SINTASSI

```
<kernel/proc.h>
proc_t *proc_reference (pid_t pid);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Il numero del processo cercato nella tabella relativa.

DESCRIZIONE

La funzione *proc_reference()* serve a produrre il puntatore all’elemento dell’array *proc_table[]* che contiene i dati del processo indicato per numero come argomento.

Viene usata dalle funzioni che non fanno parte del gruppo di ‘kernel/proc.h’.

VALORE RESTITUITO

Restituisce il puntatore all’elemento della tabella *proc_table[]* che rappresenta il processo richiesto. Se il numero del processo richiesto non può esistere, la funzione restituisce il puntatore nullo ‘**NULL**’.

FILE SORGENTI

‘kernel/proc.h’ [[94.14](#)]

'kernel/proc/proc_public.c' [94.14.5]

'kernel/proc/proc_reference.c' [94.14.6]

93.20.6 os32: proc_sch_net(9)

«

NOME

'**proc_sch_net**' - operazioni di routine relative alla gestione della rete

SINTASSI

```
<kernel/proc.h>  
void proc_sch_net (void);
```

DESCRIZIONE

La funzione *proc_sch_net()* ha il compito di scandire le interfacce di rete alla ricerca di operazioni da concludere e di pacchetti da acquisire; inoltre si occupa di aggiornare lo stato della gestione TCP e di risvegliare i processi in attesa di eventi relativi alla rete, se se ne presenta il motivo.

Questa funzione viene usata soltanto da *proc_scheduler(9)* [93.20.10], ogni volta che ci si prepara allo scambio con un altro processo.

FILE SORGENTI

'kernel/proc.h' [94.14]

'kernel/proc/proc_scheduler.c' [94.14.11]

'kernel/proc/proc_sch_net.c' [94.14.7]

VEDERE ANCHE

proc_scheduler(9) [93.20.10], *proc_sch_signals(9)* [93.20.7],
proc_sch_terminals(9) [93.20.8], *proc_sch_timers(9)* [93.20.9].

93.20.7 os32: *proc_sch_signals(9)*



NOME

‘**proc_sch_signals**’ - verifica dei segnali dei processi

SINTASSI

```
<kernel/proc.h>  
void proc_sch_signals (void);
```

DESCRIZIONE

La funzione *proc_sch_signals()* ha il compito di scandire tutti i processi della tabella *proc_table[]*, per verificare lo stato di attivazione dei segnali e procedere di conseguenza.

Dal punto di vista pratico, la funzione si limita a scandire i numeri PID possibili, demandando ad altre funzioni il compito di fare qualcosa nel caso fosse attivato l'indicatore di un segnale.

Questa funzione viene usata soltanto da *proc_scheduler(9)* [93.20.10], ogni volta che ci si prepara allo scambio con un altro processo.

FILE SORGENTI

‘kernel/proc.h’ [94.14]

‘kernel/proc/proc_scheduler.c’ [94.14.11]

‘kernel/proc/proc_sch_signals.c’ [94.14.8]

VEDERE ANCHE

proc_sig_term(9) [93.20.20], *proc_sig_core(9)* [93.20.13],
proc_sig_chld(9) [93.20.11], *proc_sig_cont(9)* [93.20.12],
proc_sig_stop(9) [93.20.19].

93.20.8 os32: *proc_sch_terminals(9)*

<<

NOME

‘**proc_sch_terminals**’ - acquisizione di un carattere dal terminale attivo

SINTASSI

```
<kernel/proc.h>  
void proc_sch_terminals (void);
```

DESCRIZIONE

La funzione *proc_sch_terminals()* ha il compito di verificare la presenza di un carattere digitato dalla console. Se c'è effettivamente un carattere digitato, dopo aver determinato a quale terminale virtuale si riferisce, lo accumula nella sua riga di inserimento.

Successivamente verifica se quel terminale virtuale è associato a un gruppo di processi; se è così e se il carattere corrisponde a **VINTR** (di norma si tratta di ciò che viene prodotto dalla combinazione di tasti [*Ctrl c*]), invia il segnale SIGINT al processo più interno del gruppo, il quale dovrebbe corrispondere presumibilmente a quello in primo piano.

Indipendentemente dal fatto che il terminale appartenga a un gruppo di processi, controlla che il carattere inserito sia stato

ottenuto, rispettivamente, con le combinazioni di tasti [*Ctrl q*], [*Ctrl r*], [*Ctrl s*] e [*Ctrl t*], nel qual caso attiva la console virtuale relativa (dalla prima alla quarta), evitando di accumulare il carattere.

Se il carattere ricevuto è tale da fare intendere la conclusione di un inserimento (per esempio il carattere *<NL>*, prodotto dalla pressione di [*Invio*]), la funzione scandisce tutti i processi sospesi in attesa di input dal terminale, risvegliandoli (ogni processo deve poi verificare se effettivamente c'è qualcosa da trarre dal terminale per sé oppure no, e se non c'è dovrebbe rimettersi in attesa).

Questa funzione viene usata soltanto da *proc_scheduler(9)* [93.20.10], ogni volta che ci si prepara allo scambio con un altro processo.

FILE SORGENTI

'kernel/proc.h' [94.14]

'kernel/proc/proc_scheduler.c' [94.14.11]

'kernel/proc/proc_sch_terminals.c' [94.14.9]

93.20.9 os32: proc_sch_timers(9)



NOME

'**proc_sch_timers**' - verifica dell'incremento del contatore del tempo

SINTASSI

```
<kernel/proc.h>
void proc_sch_timers (void);
```

DESCRIZIONE

La funzione *proc_sch_timers()* verifica che il calendario si sia incrementato di almeno una unità temporale (per os32 è un secondo soltanto) e se è così, va a risvegliare tutti i processi sospesi in attesa del passaggio di un certo tempo. Tali processi, una volta messi effettivamente in funzione, devono verificare che sia trascorsa effettivamente la quantità di tempo desiderata, altrimenti devono rimettersi a riposo in attesa del tempo rimanente.

Questa funzione viene usata soltanto da *proc_scheduler(9)* [93.20.10], ogni volta che ci si prepara allo scambio con un altro processo.

FILE SORGENTI

‘kernel/proc.h’ [94.14]

‘kernel/proc/proc_scheduler.c’ [94.14.11]

‘kernel/proc/proc_sch_timers.c’ [94.14.10]

93.20.10 os32: proc_scheduler(9)

«

NOME

‘**proc_scheduler**’ - schedulatore

SINTASSI

```
<kernel/proc.h>
extern uint32_t  _ksp;
extern uint32_t  proc_stack_pointer;
extern uint16_t  proc_stack_segment_selector;
extern pid_t     proc_current;
void proc_scheduler (void);
```

ARGOMENTI

Argomento	Descrizione
<code>extern uint32_t <i>_ksp</i>;</code>	La variabile <i>_ksp</i> contiene l'indice della pila del kernel.
<code>extern uint32_t <i>proc_stack_pointer</i>;</code>	La variabile <i>proc_stack_pointer</i> contiene l'indice della pila del processo interrotto, dal punto di vista del segmento dati del processo stesso.
<code>extern uint16_t <i>proc_stack_segment_selector</i>;</code>	La variabile <i>proc_stack_segment_selector</i> contiene il valore del registro <i>SS</i> (<i>stack segment</i>) relativo al processo sospeso.
<code>extern pid_t <i>proc_current</i>;</code>	La variabile <i>proc_current</i> contiene il numero del processo corrente, appena sospeso.

DESCRIZIONE

La funzione *proc_scheduler()* viene avviata a seguito di un'interruzione hardware, dovuta al temporizzatore, oppure a seguito di un'interruzione software, dovuta a una chiamata di sistema.

La prima cosa che fa la funzione consiste nel verificare che il valore dell'indice della pila del processo interrotto non superi lo spazio disponibile per la pila stessa. Diversamente il proces-

so viene eliminato forzatamente, con una segnalazione adeguata sul terminale attivo. Si ottiene comunque una segnalazione se l'indice si avvicina pericolosamente al limite.

Successivamente la funzione svolge delle operazioni che riguardano tutti i processi: aggiorna i contatori dei processi che attendono lo scadere di un certo tempo; verifica la presenza di segnali e predispone le azioni relative; raccoglie l'input dai terminali. Quindi aggiorna il tempo di utilizzo della CPU del processo appena interrotto.

Poi inizia la ricerca di un altro processo, candidato a essere ripreso al posto di quello interrotto. La ricerca inizia dal processo successivo a quello interrotto, senza considerare alcun criterio di precedenza. Il ciclo termina se la ricerca incontra di nuovo il processo di partenza. All'interno di questo ciclo di ricerca, se si incontra un processo pronto per essere messo in funzione, lo si scambia con quello interrotto: in pratica si salva il valore attuale dell'indice della pila, si scambiano gli stati e si aggiornano i valori di *proc_current*, *proc_stack_segment_selector* e *proc_stack_pointer*, in modo da ottenere effettivamente lo scambio all'uscita dalla funzione. Alla fine del ciclo, occorre verificare se esiste effettivamente un processo successivo attivato, perché in caso contrario, si lascia il controllo direttamente al kernel. In fine, si salva il valore accumulato in precedenza dell'indice della pila del kernel, nella variabile *_ksp*.

Questa funzione viene usata dalla routine `'irq_timer'` del file `'kernel/ibm_i386/isr.s'` e dalla funzione `sysroutine(9)` [93.20.28].

FILE SORGENTI

‘kernel/proc.h’ [94.14]

‘kernel/ibm_i386/isr.s’ [94.6.21]

‘kernel/proc/sysroutine.c’ [94.14.28]

‘kernel/proc/proc_scheduler.c’ [94.14.11]

VEDERE ANCHE

proc_sch_timers(9) [93.20.9], *proc_sch_signals(9)* [93.20.7],
proc_sch_terminals(9) [93.20.8].

93.20.11 os32: *proc_sig_chld(9)*



NOME

‘**proc_sig_chld**’ - procedura associata alla ricezione di un segnale SIGCHLD

SINTASSI

```
<kernel/proc.h>
void proc_sig_chld (pid_t parent, int sig);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>parent</i>	Numero del processo considerato, il quale potrebbe avere ricevuto un segnale SIGCHLD.
int <i>sig</i>	Numero del segnale: deve trattarsi esclusivamente di quanto corrispondente a SIGCHLD.

DESCRIZIONE

La funzione *proc_sig_chld()* si occupa di verificare che il processo specificato con il parametro *parent* abbia ricevuto precedentemente un segnale SIGCHLD. Se risulta effettivamente così, allora va a verificare se tale segnale risulta ignorato per quel processo: se è preso in considerazione verifica ancora se quel processo è sospeso proprio in attesa di un segnale SIGCHLD. Se si tratta di un processo che sta attendendo tale segnale, allora viene risvegliato, altrimenti, sempre ammesso che comunque il segnale non sia ignorato, la funzione elimina tutti i processi figli di *parent*, i quali risultano già defunti, ma non ancora rimossi dalla tabella dei processi (pertanto processi «zombie»).

In pratica, se il processo *parent* sta attendendo un segnale SIGCHLD, significa che al risveglio si aspetta di verificare la morte di uno dei suoi processi figli, in modo da poter ottenere il valore di uscita con cui questo si è concluso. Diversamente, non c'è modo di informare il processo *parent* di tali conclusioni, per cui a nulla servirebbe continuare a mantenerne le tracce nella tabella dei processi.

Questa funzione viene usata soltanto da *proc_sch_signals(9)* [93.20.7].

FILE SORGENTI

'kernel/proc.h' [94.14]

'kernel/proc/proc_sig_chld.c' [94.14.12]

VEDERE ANCHE

proc_sig_status(9) [93.20.18], *proc_sig_ignore(9)* [93.20.15],
proc_sig_off(9) [93.20.17].

93.20.12 os32: `proc_sig_cont(9)`**NOME**

‘`proc_sig_cont`’ - ripresa di un processo sospeso in attesa di qualcosa

SINTASSI

```
<kernel/proc.h>
void proc_sig_cont (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Numero del processo considerato.
int <i>sig</i>	Numero del segnale: deve trattarsi esclusivamente di quanto corrispondente a SIGCONT.

DESCRIZIONE

La funzione `proc_sig_cont()` si occupa di verificare che il processo specificato con il parametro *pid* abbia ricevuto precedentemente un segnale SIGCONT e che questo non sia stato disabilitato. In tal caso, assegna al processo lo status di «pronto» (`PROC_READY`), ammesso che non si trovasse già in questa situazione.

Lo scopo del segnale SIGCONT è quindi quello di far riprendere un processo che in precedenza fosse stato sospeso attraverso un segnale SIGSTOP, SIGTSTP, SIGTTIN oppure SIGTTOU.

Questa funzione viene usata soltanto da `proc_sch_signals(9)` [93.20.7].

FILE SORGENTI

‘kernel/proc.h’ [94.14]

‘kernel/proc/proc_sig_cont.c’ [94.14.13]

VEDERE ANCHE

proc_sig_status(9) [93.20.18], *proc_sig_ignore(9)* [93.20.15],
proc_sig_off(9) [93.20.17].

93.20.13 os32: *proc_sig_core(9)*

«

NOME

‘**proc_sig_core**’ - chiusura di un processo e scarico della memoria su file

SINTASSI

```
<kernel/proc.h>
void proc_sig_core (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Numero del processo considerato.
int <i>sig</i>	Numero del segnale: deve trattarsi di un segnale a cui si associa in modo predefinito la conclusione e lo scarico della memoria.

DESCRIZIONE

La funzione *proc_sig_core()* si occupa di verificare che il processo specificato con il parametro *pid* abbia ricevuto precedentemente un segnale tale da richiedere la conclusione e lo scarico

della memoria del processo stesso, e che il segnale in questione non sia stato disabilitato. In tal caso, la funzione chiude il processo, ma prima ne scarica la memoria su uno o due file, avvalendosi per questo della funzione *proc_dump_memory(9)* [93.20.2].

Un segnale atto a produrre lo scarico della memoria, potrebbe essere prodotto anche a seguito di un errore rilevato dalla CPU, come una divisione per zero. Tuttavia, il kernel di os32 non riesce a intrappolare errori di questo tipo, dato che dalla tabella IVT vengono presi in considerazione soltanto l'impulso del temporizzatore e le chiamate di sistema. In altri termini, se un programma produce effettivamente un errore così grave da essere rilevato dalla CPU, al sistema operativo non arriva alcuna comunicazione. Pertanto, tali segnali possono essere soltanto provocati deliberatamente.

Lo scarico della memoria, nell'eventualità di un errore così grave, dovrebbe servire per consentire un'analisi dello stato del processo nel momento del verificarsi di un errore fatale. Sotto questo aspetto, va anche considerato che l'area dati dei processi è priva di etichette che possano agevolare l'interpretazione dei contenuti e, di conseguenza, non ci sono strumenti che consentano tale attività.

Questa funzione viene usata soltanto da *proc_sch_signals(9)* [93.20.7].

FILE SORGENTI

'kernel/proc.h' [94.14]

'kernel/proc/proc_sig_core.c' [94.14.14]

VEDERE ANCHE

proc_sig_status(9) [93.20.18], *proc_sig_ignore(9)* [93.20.15],
proc_sig_off(9) [93.20.17], *proc_dump_memory(9)* [93.20.2].

93.20.14 os32: *proc_sig_handler(9)*

«

NOME

‘**proc_sig_handler**’ - attivazione di una funzione a seguito del recepimento di un segnale

SINTASSI

```
<kernel/proc.h>
void proc_sig_handler (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Numero del processo considerato.
int <i>sig</i>	Numero del segnale da verificare.

DESCRIZIONE

La funzione *proc_sig_handler()* verifica se, per un certo processo *pid*, il segnale *sig* risulti associato a una funzione. Se è così, modifica la pila dei dati del processo in modo da far sì che alla sua ripresa, venga azionata la funzione programmata, prima di riprendere con l'attività precedente. Contestualmente, il segnale *sig* viene liberato dall'associazione a una funzione.

Questa funzione viene usata da *proc_sig_cont(9)* [93.20.12], *proc_sig_core(9)* [93.20.13], *proc_sig_stop(9)* [93.20.19] e

proc_sig_term(9) [93.20.20], per verificare se un segnale sia stato associato a una funzione; prima di decidere di eseguire l'azione predefinita, in mancanza di tale associazione.

FILE SORGENTI

'kernel/proc.h' [94.14]

'kernel/proc/proc_sig_handler.c' [94.14.15]

VEDERE ANCHE

proc_sig_cont(9) [93.20.12], *proc_sig_core(9)* [93.20.13], *proc_sig_stop(9)* [93.20.19] e *proc_sig_term(9)* [93.20.20].

93.20.15 os32: *proc_sig_ignore(9)*



NOME

'**proc_sig_ignore**' - verifica dello stato di inibizione di un segnale

SINTASSI

```
<kernel/proc.h>
int proc_sig_ignore (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Numero del processo considerato.
int <i>sig</i>	Numero del segnale da verificare.

DESCRIZIONE

La funzione *proc_sig_ignore()* verifica se, per un certo processo *pid*, il segnale *sig* risulti inibito.

Questa funzione viene usata da *proc_sig_chld(9)* [93.20.11], *proc_sig_cont(9)* [93.20.12], *proc_sig_core(9)* [93.20.13], *proc_sig_stop(9)* [93.20.19] e *proc_sig_term(9)* [93.20.20], per verificare se un segnale sia stato inibito, prima di applicarne le conseguenze, nel caso fosse stato ricevuto.

VALORE RESTITUITO

Valore	Significato
1	Il segnale risulta bloccato (inibito).
0	Il segnale è abilitato regolarmente.

FILE SORGENTI

‘kernel/proc.h’ [94.14]

‘kernel/proc/proc_sig_ignore.c’ [94.14.16]

VEDERE ANCHE

proc_sig_chld(9) [93.20.11], *proc_sig_cont(9)* [93.20.12],
proc_sig_core(9) [93.20.13], *proc_sig_stop(9)* [93.20.19]m
proc_sig_term(9) [93.20.20].

93.20.16 os32: *proc_sig_off(9)*

«

Vedere *proc_sig_on(9)* [93.20.17].

93.20.17 os32: *proc_sig_on(9)*

«

NOME

‘**proc_sig_on**’, ‘**proc_sig_off**’ - registrazione o cancellazione di un segnale per un processo

SINTASSI

```
<kernel/proc.h>
void proc_sig_on (pid_t pid, int sig);
void proc_sig_off (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Numero del processo considerato.
int <i>sig</i>	Numero del segnale da registrare o da cancellare.

DESCRIZIONE

La funzione *proc_sig_on()* annota per il processo *pid* la ricezione del segnale *sig*; la funzione *proc_sig_off()* procede invece in senso opposto, cancellando quel segnale.

La funzione *proc_sig_off()* viene usata quando l'azione prevista per un segnale che risulta ricevuto è stata eseguita, allo scopo di riportare l'indicatore di quel segnale in una condizione di riposo. Si tratta delle funzioni *proc_sig_chld(9)* [93.20.11], *proc_sig_cont(9)* [93.20.12], *proc_sig_core(9)* [93.20.13], *proc_sig_stop(9)* [93.20.19] e *proc_sig_term(9)* [93.20.20].

La funzione *proc_sig_on()* viene usata quando risulta acquisito un segnale o quando il contesto lo deve produrre, per annotarlo. Si tratta delle funzioni *s__exit(9)* [93.12] e *s_kill(9)* [93.12].

FILE SORGENTI

'kernel/proc.h' [94.14]

'kernel/proc/proc_sig_on.c' [94.14.18]

‘kernel/proc/proc_sig_off.c’ [94.14.17]

VEDERE ANCHE

proc_sig_chld(9) [93.20.11], *proc_sig_cont(9)* [93.20.12],
proc_sig_core(9) [93.20.13], *proc_sig_stop(9)* [93.20.19],
proc_sig_term(9) [93.20.20].

93.20.18 os32: *proc_sig_status(9)*

«

NOME

‘*proc_sig_status*’ - verifica dello stato di ricezione di un segnale

SINTASSI

```
<kernel/proc.h>
int proc_sig_status (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Numero del processo considerato.
int <i>sig</i>	Numero del segnale da verificare.

DESCRIZIONE

La funzione *proc_sig_status()* verifica se, per un certo processo *pid*, il segnale *sig* risulti essere stato ricevuto (registrato).

Questa funzione viene usata da *proc_sig_chld(9)* [93.20.11], *proc_sig_cont(9)* [93.20.12], *proc_sig_core(9)* [93.20.13], *proc_sig_stop(9)* [93.20.19] e *proc_sig_term(9)* [93.20.20], per

verificare se un segnale è stato ricevuto effettivamente, prima di applicarne eventualmente le conseguenze.

VALORE RESTITUITO

Valore	Significato
1	Il segnale risulta ricevuto.
0	Il segnale risulta cancellato.

FILE SORGENTI

‘kernel/proc.h’ [94.14]

‘kernel/proc/proc_sig_status.c’ [94.14.19]

VEDERE ANCHE

proc_sig_chld(9) [93.20.11], *proc_sig_cont(9)* [93.20.12],
proc_sig_core(9) [93.20.13], *proc_sig_stop(9)* [93.20.19],
proc_sig_term(9) [93.20.20].

93.20.19 os32: *proc_sig_stop(9)*



NOME

‘**proc_sig_stop**’ - sospensione di un processo

SINTASSI

```
<kernel/proc.h>
void proc_sig_stop (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
<code>pid_t <i>pid</i></code>	Numero del processo considerato.
<code>int <i>sig</i></code>	Numero del segnale: deve trattarsi di SIGSTOP, SIGTSTP, SIGTTIN o SIGTTOU.

DESCRIZIONE

La funzione *proc_sig_stop()* si occupa di verificare che il processo specificato con il parametro *pid* abbia ricevuto precedentemente un segnale SIGSTOP, SIGTSTP, SIGTTIN o SIGTTOU, e che questo non sia stato disabilitato. In tal caso, sospende il processo, lasciandolo in attesa di un segnale (SIGCONT).

Questa funzione viene usata soltanto da *proc_sch_signals(9)* [93.20.7].

FILE SORGENTI

‘kernel/proc.h’ [94.14]

‘kernel/proc/proc_sig_stop.c’ [94.14.20]

VEDERE ANCHE

proc_sig_status(9) [93.20.18], *proc_sig_ignore(9)* [93.20.15],
proc_sig_off(9) [93.20.17].

93.20.20 os32: *proc_sig_term(9)*

«

NOME

‘*proc_sig_term*’ - conclusione di un processo

SINTASSI

```
<kernel/proc.h>
void proc_sig_term (pid_t pid, int sig);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>pid</i>	Numero del processo considerato.
int <i>sig</i>	Numero del segnale: deve trattarsi di un segnale per cui si associa la conclusione del processo, ma senza lo scarico della memoria.

DESCRIZIONE

La funzione *proc_sig_term()* si occupa di verificare che il processo specificato con il parametro *pid* abbia ricevuto precedentemente un segnale per cui si prevede generalmente la conclusione del processo. Inoltre, la funzione verifica che il segnale non sia stato inibito, con l'eccezione che per il segnale SIGKILL un'eventuale inibizione non viene considerata (in quanto segnale non mascherabile). Se il segnale risulta ricevuto e valido, procede con la conclusione del processo.

Questa funzione viene usata soltanto da *proc_sch_signals(9)* [93.20.7].

FILE SORGENTI

'kernel/proc.h' [94.14]

'kernel/proc/proc_sig_term.c' [94.14.21]

VEDERE ANCHE

proc_sig_status(9) [93.20.18], *proc_sig_ignore(9)* [93.20.15],
proc_sig_off(9) [93.20.17].

93.20.21 os32: *proc_sys_exec(9)*

«

NOME

‘**proc_sys_exec**’ - sostituzione di un processo esistente con un altro, ottenuto dal caricamento di un file eseguibile

SINTASSI

```
<kernel/proc.h>
```

```
int proc_sys_exec (pid_t pid, const char *path,  
                  unsigned int argc, char *arg_data,  
                  unsigned int envc, char *env_data)
```

ARGOMENTI

Argomento	Descrizione
<code>pid_t <i>pid</i></code>	Il numero del processo corrispondente.
<code>const char *<i>path</i></code>	Il percorso assoluto del file da caricare ed eseguire.
<code>unsigned int <i>argc</i></code>	La quantità di argomenti per l'avvio del nuovo processo, incluso il nome del processo stesso.
<code>char *<i>arg_data</i></code>	Una sequenza di stringhe (dopo la terminazione di una inizia la successiva), ognuna contenente un argomento da passare al processo.
<code>unsigned int <i>envc</i></code>	La quantità di variabili di ambiente da passare al nuovo processo.
<code>char *<i>env_data</i></code>	Una sequenza di stringhe (dopo la terminazione di una inizia la successiva), ognuna contenente l'assegnamento di una variabile di ambiente.

I parametri *arg_data* e *env_data* sono stringhe multiple, nel senso che sono separate le une dalle altre dal codice nullo di terminazione. Per sapere quante sono effettivamente le stringhe da cercare a partire dai puntatori che costituiscono effettivamente questi due parametri, si usano *argc* e *envc*.

DESCRIZIONE

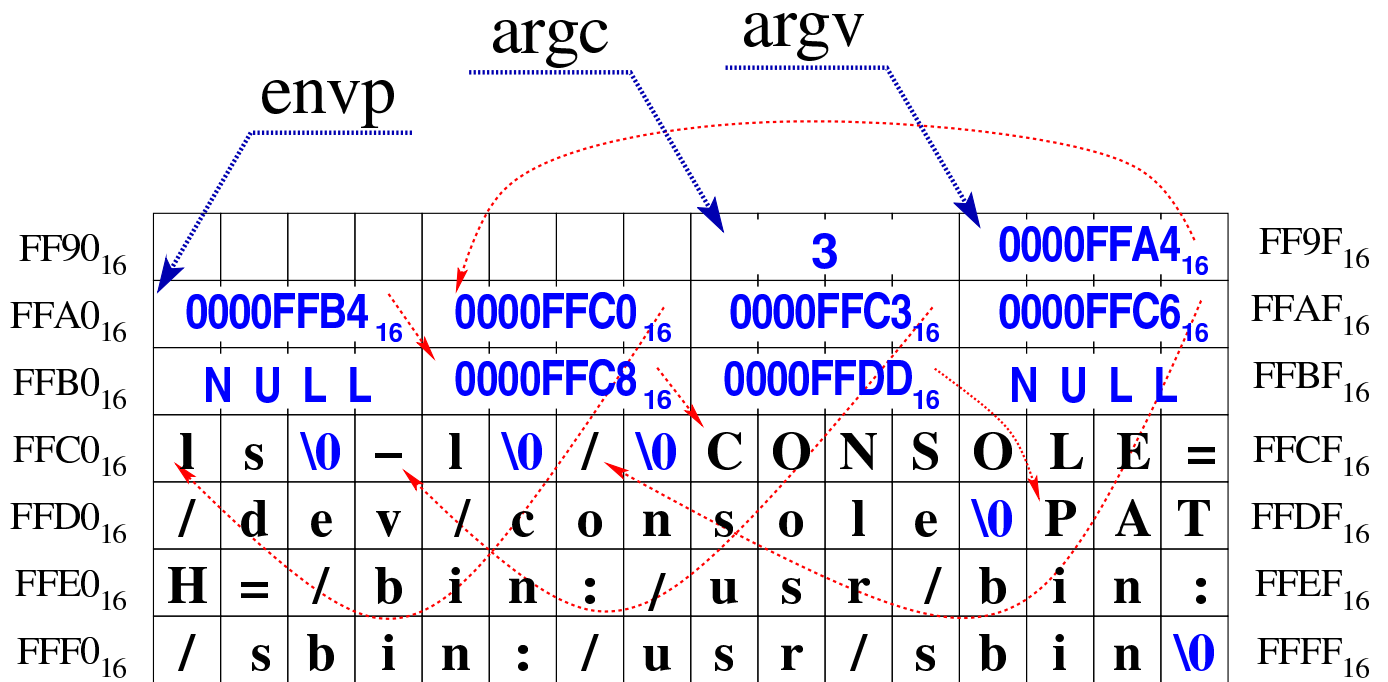
La funzione *proc_sys_exec()* serve a mettere in pratica la chiamata di sistema *execve(2)* [87.14], destinata a rimpiazzare il processo in corso con un nuovo processo, caricato da un file eseguibile.

La funzione *proc_sys_exec()*, dopo aver verificato che si tratti

effettivamente di un file eseguibile valido e che ci siano i permessi per metterlo in funzione, procede all'allocazione della memoria, quindi legge il file e copia opportunamente le componenti di questo nelle aree di memoria allocate.

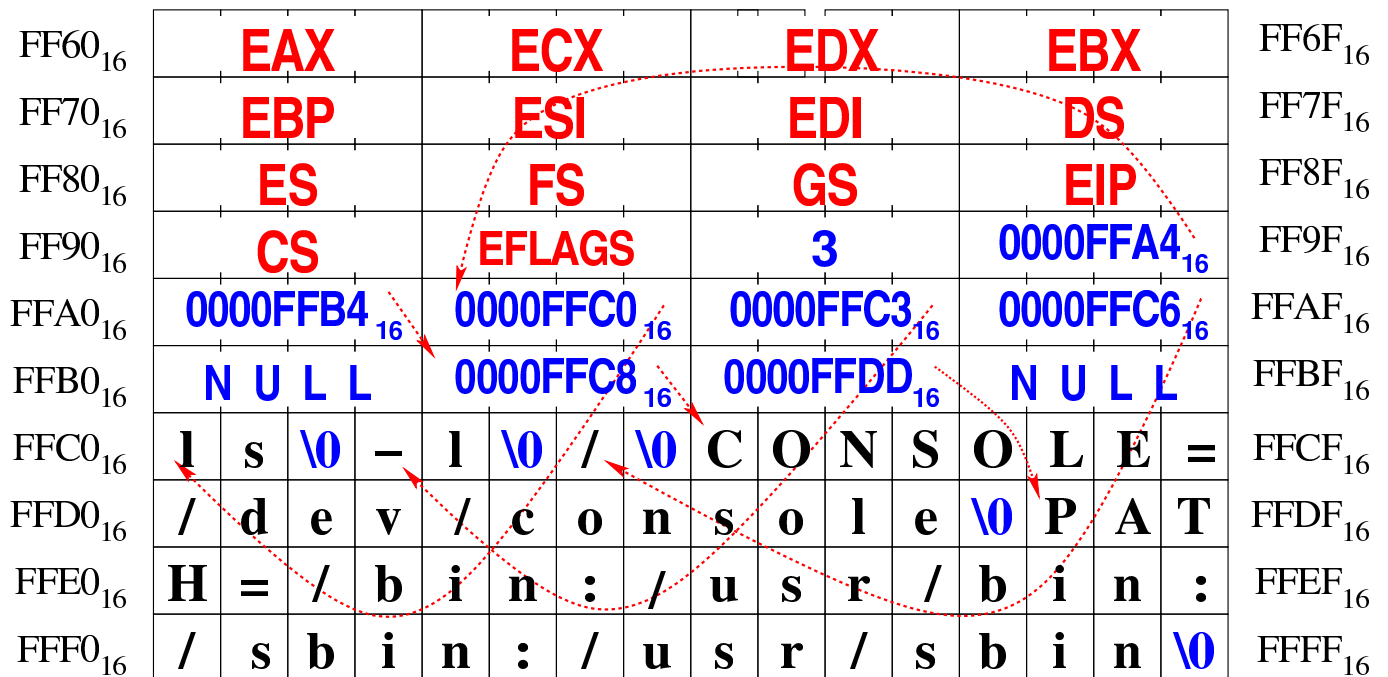
Terminato il caricamento del file, viene ricostruita in memoria la pila dei dati del nuovo processo. Prima si mettono sul fondo le stringhe delle variabili di ambiente e quelle degli argomenti della chiamata, quindi si aggiungono i puntatori alle stringhe delle variabili di ambiente, ricostruendo così l'array noto convenzionalmente come '**envp**[]', continuando con l'aggiunta dei puntatori alle stringhe degli argomenti della chiamata, per riprodurre l'array '**argv**[]'. Per ricostruire gli argomenti della chiamata della funzione *main()* dell'applicazione, vanno però aggiunti ancora: il puntatore all'inizio dell'array delle stringhe che descrivono le variabili di ambiente, il puntatore all'array delle stringhe che descrivono gli argomenti della chiamata e il valore che rappresenta la quantità di argomenti della chiamata.

Figura 84.69. Caricamento degli argomenti della chiamata della funzione *main()*.



Fatto ciò, vanno aggiunti tutti i valori necessari allo scambio dei processi, costituiti dai vari registri da rimpiazzare.

Figura 84.70. Completamento della pila con i valori dei registri.



Superato il problema della ricostruzione della pila dei dati, la funzione *proc_sys_exec()* predispone i descrittori di standard input, standard output e standard error, quindi libera la memoria usata dal processo chiamante e ne rimpiazza i dati nella tabella dei processi con quelli del nuovo processo caricato.

Questa funzione viene usata soltanto da *sysroutine(9)* [93.20.28], in occasione del ricevimento di una chiamata di sistema di tipo **'SYS_EXEC'**.

FILE SORGENTI

'lib/unistd/execve.c' [95.30.14]
'lib/sys/os32/sys.s' [95.21.7]
'kernel/proc.h' [94.14]
'kernel/ibm_i386/isr.s' [94.6.21]
'kernel/proc/sysroutine.c' [94.14.28]
'kernel/proc/proc_sys_exec.c' [94.14.22]

VEDERE ANCHE

execve(2) [87.14], *sys(2)* [87.56], *sysroutine(9)* [93.20.28],
proc_scheduler(9) [93.20.10], *path_inode(9)* [93.6.41],
inode_check(9) [93.6.8], *inode_put(9)* [93.6.20],
inode_file_read(9) [93.6.10], *dev_io(9)* [93.4.1].

93.20.22 os32: *proc_timer_init(9)*

«

NOME

'proc_timer_init' - inizializzazione del generatore di impulsi (temporizzatore)

SINTASSI

```
<kernel/proc.h>
void proc_timer_init (clock_t freq);
```

ARGOMENTI

Argomento	Descrizione
clock_t <i>freq</i>	Frequenza in Hz che si intende produrre. Tuttavia, tutto il sistema è organizzato per gestire una frequenza di 100 Hz, il quale diventa praticamente obbligatorio.

DESCRIZIONE

La funzione *proc_timer_init()* viene chiamata esclusivamente dalla funzione *proc_init(9)* [93.20.3], per inizializzare il generatore di impulsi alla frequenza di **CLOCKS_PER_SEC** Hz, pari a 100 Hz.

FILE SORGENTI

‘kernel/proc.h’ [94.14]

‘kernel/proc/proc_timer_init.c’ [94.14.23]

VEDERE ANCHE

kmain(9) [93.13], *proc_init(9)* [93.20.3].

93.20.23 os32: *proc_wakeup(9)*

«

NOME

‘*proc_wakeup_pipe_read*’, ‘*proc_wakeup_pipe_write*’, ‘*proc_wakeup_terminal*’ - risveglio dei processi in attesa di un condotto o di un terminale.

SINTASSI

```
<kernel/proc.h>
void proc_wakeup_pipe_read (inode_t *inode);
void proc_wakeup_pipe_write (inode_t *inode);
void proc_wakeup_terminal (void);
```

ARGOMENTI

Argomento	Descrizione
inode_t * <i>inode</i>	Riferimento a una voce della tabella degli inode, con cui si identifica il condotto per il quale si attende di poter leggere o scrivere.

DESCRIZIONE

Le funzioni *proc_wakeup_...()* hanno lo scopo di scandire la tabella dei processi, alla ricerca di quelli da risvegliare, a seguito di un evento che richiede o può richiedere la loro attenzione. Le funzioni *proc_wakeup_pipe_...()* risvegliano esclusivamente i processi che sono in attesa di accedere a un condotto identificato attraverso un inode, mentre *proc_wakeup_terminal()* va a risvegliare tutti i processi in attesa del terminale, anche quelli che probabilmente non sono interessati direttamente da un input disponibile da tastiera.

FILE SORGENTI

‘kernel/proc.h’ [94.14]

‘kernel/proc/proc_wakeup_pipe_read.c’ [94.14.24]

‘kernel/proc/proc_wakeup_pipe_write.c’ [94.14.25]

‘kernel/proc/proc_wakeup_terminal.c’ [94.14.26]

VEDERE ANCHE

proc_sch_terminals(9) [93.20.8].

93.20.24 os32: *proc_wakeup_pipe_read(9)*

Vedere *proc_wakeup(9)* [93.20.23].

93.20.25 os32: *proc_wakeup_pipe_write(9)*

Vedere *proc_wakeup(9)* [93.20.23].

93.20.26 os32: *proc_wakeup_terminal(9)*

Vedere *proc_wakeup(9)* [93.20.23].

93.20.27 os32: *ptr(9)*

NOME

‘**ptr**’ - conversione di un puntatore relativo all’area dati di un processo in un puntatore valido per il kernel

SINTASSI

```
<kernel/proc.h>
void *ptr (pid_t pid, void *p);
```

DESCRIZIONE

La funzione *ptr()* ha il compito di convertire un puntatore proveniente dall’area dati di un processo in un puntatore valido dal punto di vista del kernel. In pratica, il puntatore corrispondente al parametro *p* va inteso come riferito a un certo processo *pid*; la funzione restituisce un puntatore equivalente, ma valido per il kernel.

FILE SORGENTI

‘kernel/proc.h’ [94.14]

‘kernel/proc/ptr.c’ [94.14.27]

VEDERE ANCHE

sysroutine(9) [93.20.28].

93.20.28 os32: *sysroutine(9)*

«

NOME

‘**s_sysroutine**’ - attuazione delle chiamate di sistema

SINTASSI

```
<kernel/proc.h>
extern pid_t proc_current;
void sysroutine (uint32_t syscallnr, uint32_t msg_off,
                 uint32_t msg_size);
```

ARGOMENTI

Argomento	Descrizione
pid_t <i>proc_current</i>	Il numero del processo interrotto.
uint32_t <i>syscallnr</i>	Il numero della chiamata di sistema.
uint32_t <i>msg_off</i>	Nonostante il tipo di variabile, si tratta del puntatore alla posizione di memoria in cui inizia il messaggio con gli argomenti della chiamata di sistema, ma tale puntatore è valido solo nell’ambito del segmento dati del processo interrotto.
uint32_t <i>msg_size</i>	La lunghezza del messaggio della chiamata di sistema.

DESCRIZIONE

La funzione *sysroutine()* viene chiamata esclusivamente dalla routine `'isr_128'`, contenuta nel file `'kernel/ibm_i386/isr.s'` [94.6.21], a seguito di una chiamata di sistema.

Inizialmente, la funzione individua l'indirizzo corrispondente alla posizione del messaggio proveniente dal processo, dal punto di vista del kernel; in pratica traduce *msg_off* in qualcosa di adatto al kernel.

Attraverso un'unione di variabili strutturate, tutti i tipi di messaggi gestibili per le chiamate di sistema vengono dichiarati assieme in un'unica area di memoria. Successivamente, la funzione deve trasferire il messaggio, dall'indirizzo calcolato precedentemente all'inizio dell'unione in questione.

Quando la funzione è in grado di accedere ai dati del messaggio, procede con una grande struttura di selezione, sulla base del tipo di messaggio, quindi esegue ciò che è richiesto, avvalendosi prevalentemente di altre funzioni, interpretando il messaggio in modo diverso a seconda del tipo di chiamata.

Il messaggio originale viene poi sovrascritto con le informazioni prodotte dall'azione richiesta, in particolare viene trasferito anche il valore della variabile *errno* del kernel, in modo che possa essere recepita anche dal processo che ha eseguito la chiamata, in caso di esito erraneo. Pertanto, il messaggio viene riscritto a partire dall'indirizzo da cui era stato copiato precedentemente, in modo da renderlo disponibile effettivamente al processo chiamante.

Quando la funzione *sysroutine()* ha finito il suo lavoro, chiama a sua volta *proc_scheduler(9)* [93.20.10], perché con l'occasione

provveda eventualmente alla sostituzione del processo attivo con un altro che si trovi nello stato di pronto.

VALORE RESTITUITO

La funzione non restituisce alcun valore, in quanto tutto ciò che c'è da restituire viene trasmesso con la riscrittura del messaggio, nell'area di memoria originale.

FILE SORGENTI

'lib/sys/os32/sys.s' [95.21.7]

'kernel/proc.h' [94.14]

'kernel/ibm_i386/isr.s' [94.6.21]

'kernel/proc/sysroutine.c' [94.14.28]

VEDERE ANCHE

sys(2) [87.56], *proc_scheduler(9)* [93.20.10], *dev_io(9)* [93.4.1], *lib_s(9)* [93.12].

93.21 os32: route(9)

«

Il file 'kernel/net/route.h' [94.12.33] descrive le funzioni per la gestione degli instradamenti IPv4 secondo os32.

Per la descrizione sulla gestione degli instradamenti IPv4 secondo il sistema os32, si rimanda alla sezione 84.10.3. La tabella successiva che sintetizza l'uso delle funzioni di questo gruppo, è tratta da lì.

Tabella 84.128. Funzioni per la gestione della tabella degli instradamenti.

Funzione	Descrizione
<pre>void route_init (void);</pre>	<p>Inizializza la tabella <i>route_table[]</i>, predisponendo la voce <i>route_table[0]</i> per l'interfaccia locale <i>loopback</i> [94.12.34].</p>
<pre>void route_sort (void);</pre>	<p>Riordina la tabella degli instradamenti [94.12.39].</p>
<pre>h_addr_t route_remote_to_local (h_addr_t <i>remote</i>);</pre>	<p>Restituisce l'indirizzo IPv4 locale, più adatto per intrattenere una connessione con l'indirizzo remoto fornito come argomento. Questo tipo di analisi viene determinato partendo dalla tabella degli instradamenti, per determinare l'indirizzo IPv4 locale dell'interfaccia interessata dal collegamento [94.12.37].</p>

Funzione	Descrizione
<pre> h_addr_t route_remote_to_router (h_addr_t <i>remote</i>); </pre>	<p>Restituisce l'indirizzo IPv4 del router da utilizzare per raggiungere l'indirizzo IPv4 remoto specificato. Se l'indirizzo restituito è pari a -1 significa che non è stata ottenuta alcuna voce corrispondente, mentre se si ottiene zero significa che non c'è bisogno di router per raggiungere la destinazione [94.12.38].</p>

93.22 os32: screen(9)



Il file 'kernel/driver/screen.h' [94.4.30] descrive le funzioni per la gestione dello schermo, in relazione alla gestione complessiva dei terminali virtuali.

Per la descrizione dell'organizzazione della gestione dello schermo di os32, si rimanda alla sezione 84.7.5.2. La tabella successiva che sintetizza l'uso delle funzioni di questo gruppo, è tratta da lì.

Tabella 84.89. Funzioni per la gestione dello schermo, dichiarate nel file di intestazione 'kernel/driver/screen.h' e descritte nei file contenuti nella directory 'kernel/driver/screen/'.

Funzione	Descrizione
<pre>int screen_clear (screen_t *<i>screen</i>);</pre>	<p>Ripulisce il contenuto dello schermo selezionato, riposizionando il cursore all'inizio.</p>
<pre>screen_t *screen_current (void);</pre>	<p>Restituisce il puntatore alla voce della tabella <i>screen_table[]</i> che descrive lo schermo virtuale attivo.</p>
<pre>void screen_init (void);</pre>	<p>Inizializza la gestione degli schermi virtuali, ripulendoli e collocando il cursore all'inizio. Questa funzione viene usata da <i>tty_init()</i>.</p>
<pre>int screen_newline (screen_t *<i>screen</i>);</pre>	<p>Produce sullo schermo virtuale selezionato un avanzamento alla riga successiva. Ciò può comportare semplicemente il riposizionamento del cursore, oppure lo scorrimento in avanti del contenuto, quando il cursore si trova già sull'ultima riga visualizzabile.</p>

Funzione	Descrizione
<pre>int screen_number (screen_t *<i>screen</i>);</pre>	<p>Restituisce il numero dello schermo corrispondente al puntatore fornito, purché questo sia valido. È in pratica l'opposto della funzione <i>screen_pointer()</i>.</p>
<pre>screen_t *screen_pointer (int <i>scrn</i>);</pre>	<p>Restituisce il puntatore alla voce della tabella <i>screen_table[]</i> che descrive lo schermo virtuale indicato per numero. È in pratica l'opposto della funzione <i>screen_number()</i>.</p>
<pre>int screen_putc (screen_t *<i>screen</i>, int <i>c</i>);</pre>	<p>Colloca sullo schermo virtuale individuato dal puntatore che costituisce il primo parametro, il carattere richiesto come secondo. Se il carattere in questione è <CR> o <LF>, si produce un avanzamento alla riga successiva, mentre con un carattere <BS> si produce un arretramento del cursore.</p>

Funzione	Descrizione
<pre>uint16_t screen_cell (c, <i>attributo</i>);</pre>	<p>Si tratta di una macroistruzione che produce il valore corretto per una cella dello schermo VGA, contenente sia l'informazione sul carattere, sia quella dell'attributo associato.</p>
<pre>int screen_scroll (screen_t *<i>screen</i>);</pre>	<p>Fa scorrere in avanti lo schermo, di una riga, ricollocando di conseguenza il cursore.</p>
<pre>int screen_select (screen_t *<i>screen</i>);</pre>	<p>Seleziona lo schermo indicato come schermo attivo, facendone apparire il contenuto sullo schermo VGA reale.</p>
<pre>void screen_update (screen_t *<i>screen</i>);</pre>	<p>Aggiorna la memoria VGA sulla base della copia che rappresenta lo schermo virtuale attivo. L'aggiornamento implica anche la collocazione del cursore visibile in corrispondenza della posizione attuale.</p>

93.23 os32: tcp(9)

I file 'kernel/net/tcp.h' [94.12.40] e 'kernel/net/udp.h' [94.12.53] descrivono le funzioni per la gestione dei protocolli TCP e UDP, secondo os32.



Per la descrizione sulla gestione dei protocolli TCP e UDP da parte di os32, si rimanda alla sezione [84.12](#). La tabella successiva che sintetizza l'uso delle funzioni di questo gruppo, è tratta da lì.

Tabella 84.132. Funzioni per la gestione dei protocolli TCP e UDP.

Funzione	Descrizione
<pre>int tcp_tx_raw (h_port_t <i>sport</i>, h_port_t <i>dport</i>, uint32_t <i>seq</i>, uint32_t <i>ack_seq</i>, int <i>flags</i>, h_addr_t <i>saddr</i>, h_addr_t <i>daddr</i>, const void *<i>buffer</i>, size_t <i>size</i>);</pre>	<p>Costruisce un pacchetto TCP, utilizzando i dati forniti come argomenti della chiamata; quindi lo trasmette attraverso la funzione <i>ip_tx()</i> che a sua volta provvede a imbustarlo in un pacchetto IP prima della trasmissione effettiva. Si tratta comunque di una funzione usata soltanto per fare dei test di funzionamento [94.12.50].</p>
<pre>void tcp_show (h_addr_t <i>src</i>, h_addr_t <i>dst</i>, const struct tcphdr *<i>tcphdr</i>);</pre>	<p>Funzione diagnostica realizzata per visualizzare alcune informazioni su un pacchetto TCP, di cui si conosce il contenuto e gli indirizzi IPv4. Questa funzione viene usata prevalentemente da <i>tcp()</i>, quando si attiva la macro-variabile DEBUG [94.12.46].</p>

Funzione	Descrizione
<pre>int tcp_tx_rst (void *<i>ip_packet</i>);</pre>	Sulla base di un pacchetto IP ricevuto con un contenuto TCP, trasmette un pacchetto TCP di azzera-mento (RST) [94.12.51].
<pre>int tcp_tx_sock (void *<i>sock_item</i>);</pre>	Trasmette quanto conte-nuto nella coda del socket indicato come argomento, ammesso che il socket sia nella condizione di poter trasmettere [94.12.52].
<pre>int tcp_tx_ack (void *<i>sock_item</i>);</pre>	Trasmette un pacchetto TCP vuoto contenente la conferma di quanto rice-vuto in precedenza, sul-la base dello stato attuale del socket indicato come argomento [94.12.49].
<pre>int tcp_rx_ack (void *<i>sock_item</i>, void *<i>packet</i>);</pre>	Verifica che il pacchet-to indicato come secondo parametro, contenga una conferma valida per il soc-cket specificato come pri-mo parametro [94.12.44].

Funzione	Descrizione
<pre>int tcp_rx_data (void *<i>sock_item</i>, void *<i>packet</i>);</pre>	Legge il contenuto di un pacchetto TCP e lo copia all'interno della memoria tampone del socket rappresentata da <i>sock_table[s].tcp.recv_data</i> (dove <i>s</i> è l'indice del socket considerato) [94.12.45].
<pre>int tcp_connect (void *<i>sock_item</i>);</pre>	Fa in modo di mettere il socket in connessione, ammesso che ciò sia possibile. Questa funzione serve a <i>s_connect()</i> [94.12.43].
<pre>int tcp_close (void *<i>sock_item</i>);</pre>	Fa in modo di mettere il socket nello stato di chiusura, ammesso che ciò sia possibile. Questa funzione serve a <i>s_close()</i> [94.12.42].
<pre>int tcp (void);</pre>	Viene chiamata da <i>proc_sch_net()</i> e si occupa di gestire lo stato di tutte le connessioni TCP in essere in quel momento [94.12.41].

Funzione	Descrizione
<pre>int udp_tx (h_port_t <i>sport</i>, h_port_t <i>dport</i>, h_addr_t <i>saddr</i>, h_addr_t <i>daddr</i>, const void *<i>buffer</i>, size_t <i>size</i>);</pre>	<p>Assembla un pacchetto UDP e lo trasmette (dopo aver costruito a sua volta il pacchetto IPv4 complessivo) [94.12.54].</p>

93.24 os32: tty(9)

Il file ‘kernel/driver/tty.h’ [94.4.42] descrive le funzioni per la gestione dei terminali virtuali. «

Per la descrizione dell’organizzazione della gestione dei terminali virtuali di os32, si rimanda alla sezione 84.7.5. La tabella successiva che sintetizza l’uso delle funzioni di questo gruppo, è tratta da lì.

Tabella 84.85. Funzioni per l’accesso al terminale, dichiarate nel file di intestazione ‘kernel/driver/tty.h’ e descritte nei file contenuti nella directory ‘kernel/driver/tty/’.

Funzione	Descrizione
<pre>dev_t tty_console (dev_t <i>device</i>);</pre>	Seleziona un terminale virtuale, rendendolo attivo, specificandone il numero del dispositivo. La funzione restituisce il dispositivo attivo in precedenza e se le viene fornito solo il valore zero, il terminale virtuale non cambia, ma si ottiene comunque di conoscere qual è quello attuale.
<pre>void tty_init (void);</pre>	Inizializza la gestione dei terminali virtuali, popolando anche la tabella <i>tty_table[]</i> con i valori predefiniti. Questa funzione viene usata una volta sola all'interno di <i>kmain()</i> .
<pre>int tty_read (dev_t <i>device</i>);</pre>	Legge un carattere dal terminale virtuale specificato attraverso il numero di dispositivo. La lettura avviene solo se l'input da tastiera risulta concluso, altrimenti la funzione restituisce il valore -1.

Funzione	Descrizione
<pre>tty_t *tty_reference (dev_t <i>device</i>);</pre>	Restituisce il puntatore alla voce della tabella <i>tty_table[]</i> contenente le informazioni sul terminale virtuale indicato attraverso il numero di dispositivo. Se il dispositivo indicato non è valido, si ottiene il puntatore nullo; se viene richiesto il dispositivo indefinito, si ottiene il puntatore all'inizio della tabella.
<pre>void tty_write (dev_t <i>device</i>, int <i>c</i>);</pre>	Scrive un carattere sullo schermo del terminale specificato.



94	Script e sorgenti del kernel	967
94.1	os32: directory principale	980
94.2	os32: «kernel/blk.h»	1008
94.3	os32: «kernel/dev.h»	1017
94.4	os32: «kernel/dm.h»	1037
94.5	os32: «kernel/fs.h»	1147
94.6	os32: «kernel/ibm_i386.h»	1277
94.7	os32: «kernel/lib_k.h»	1332
94.8	os32: «kernel/lib_s.h»	1341
94.9	os32: «kernel/main.h»	1505
94.10	os32: «kernel/memory.h»	1523
94.11	os32: «kernel/multiboot.h»	1539
94.12	os32: «kernel/net.h»	1544
94.13	os32: «kernel/part.h»	1685
94.14	os32: «kernel/proc.h»	1686
95	Sorgenti della libreria generale	1789
95.1	os32: file isolati della directory «lib/»	1799
95.2	os32: «lib/_gcc.h»	1811
95.3	os32: «lib/arpa/inet.h»	1818
95.4	os32: «lib/dirent.h»	1825
95.5	os32: «lib/errno.h»	1835

95.6	os32: «lib/fcntl.h»	1846
95.7	os32: «lib/grp.h»	1852
95.8	os32: «lib/inttypes.h»	1857
95.9	os32: «lib/libgen.h»	1865
95.10	os32: «lib/netinet/icmp.h»	1870
95.11	os32: «lib/netinet/in.h»	1874
95.12	os32: «lib/netinet/ip.h»	1877
95.13	os32: «lib/netinet/tcp.h»	1879
95.14	os32: «lib/netinet/udp.h»	1882
95.15	os32: «lib/pwd.h»	1883
95.16	os32: «lib/setjmp.h»	1887
95.17	os32: «lib/signal.h»	1891
95.18	os32: «lib/stdio.h»	1897
95.19	os32: «lib/stdlib.h»	2013
95.20	os32: «lib/string.h»	2067
95.21	os32: «lib/sys/os32.h»	2091
95.22	os32: «lib/sys/sa_family_t.h»	2128
95.23	os32: «lib/sys/socket.h»	2129
95.24	os32: «lib/sys/socklen_t.h»	2143
95.25	os32: «lib/sys/stat.h»	2144
95.26	os32: «lib/sys/types.h»	2154
95.27	os32: «lib/sys/wait.h»	2156
95.28	os32: «lib/termios.h»	2158
95.29	os32: «lib/time.h»	2162
95.30	os32: «lib/unistd.h»	2177

95.31	os32: «lib/utime.h»	2228
96	Sorgenti delle applicazioni	2231
96.1	os32: directory «applic/»	2233
	Indice analitico del volume	2481

Script e sorgenti del kernel



94.1	os32: directory principale	980
94.1.1	applic.sep.ld	980
94.1.2	bochs	981
94.1.3	elf-to-os32	982
94.1.4	fdisk	985
94.1.5	file_image_functions	986
94.1.6	format	993
94.1.7	kernel.ld	994
94.1.8	makeit.sep	995
94.1.9	qemu	1006
94.1.10	syslinux	1007
94.1.11	tap0	1008
94.2	os32: «kernel/blk.h»	1008
94.2.1	kernel/blk/blk_ata.c	1010
94.2.2	kernel/blk/blk_cache_check.c	1012
94.2.3	kernel/blk/blk_cache_init.c	1013
94.2.4	kernel/blk/blk_cache_read.c	1014
94.2.5	kernel/blk/blk_cache_save.c	1015
94.2.6	kernel/blk/blk_public.c	1017
94.3	os32: «kernel/dev.h»	1017
94.3.1	kernel/dev/dev_ata.c	1019
94.3.2	kernel/dev/dev_dm.c	1022

94.3.3	kernel/dev/dev_io.c	1023
94.3.4	kernel/dev/dev_kmem.c	1024
94.3.5	kernel/dev/dev_mem.c	1031
94.3.6	kernel/dev/dev_tty.c	1034
94.4	os32: «kernel/dm.h»	1037
94.4.1	kernel/dm/dm_init.c	1041
94.4.2	kernel/dm/dm_public.c	1044
94.4.3	kernel/driver/ata.h	1044
94.4.4	kernel/driver/ata/ata_cmd_identify_device.c	1049
94.4.5	kernel/driver/ata/ata_cmd_read_sectors.c	1051
94.4.6	kernel/driver/ata/ata_cmd_write_sectors.c	1053
94.4.7	kernel/driver/ata/ata_device.c	1055
94.4.8	kernel/driver/ata/ata_drq.c	1057
94.4.9	kernel/driver/ata/ata_init.c	1059
94.4.10	kernel/driver/ata/ata_lba28.c	1066
94.4.11	kernel/driver/ata/ata_public.c	1067
94.4.12	kernel/driver/ata/ata_rdy.c	1068
94.4.13	kernel/driver/ata/ata_reset.c	1070
94.4.14	kernel/driver/ata/ata_valid.c	1070
94.4.15	kernel/driver/kbd.h	1071
94.4.16	kernel/driver/kbd/kbd_isr.c	1072
94.4.17	kernel/driver/kbd/kbd_load.c	1077
94.4.18	kernel/driver/kbd/kbd_public.c	1081
94.4.19	kernel/driver/nic/ne2k.h	1081
94.4.20	kernel/driver/nic/ne2k/ne2k_check.c	1085

94.4.21	kernel/driver/nic/ne2k/ne2k_isr.c	1088
94.4.22	kernel/driver/nic/ne2k/ne2k_isr_expect.c	1090
94.4.23	kernel/driver/nic/ne2k/ne2k_reset.c	1092
94.4.24	kernel/driver/nic/ne2k/ne2k_rx.c	1104
94.4.25	kernel/driver/nic/ne2k/ne2k_rx_reset.c	1113
94.4.26	kernel/driver/nic/ne2k/ne2k_tx.c	1115
94.4.27	kernel/driver/pci.h	1119
94.4.28	kernel/driver/pci/pci_init.c	1122
94.4.29	kernel/driver/pci/pci_public.c	1125
94.4.30	kernel/driver/screen.h	1125
94.4.31	kernel/driver/screen/screen_clear.c	1127
94.4.32	kernel/driver/screen/screen_current.c	1128
94.4.33	kernel/driver/screen/screen_init.c	1128
94.4.34	kernel/driver/screen/screen_new_line.c	1129
94.4.35	kernel/driver/screen/screen_number.c	1130
94.4.36	kernel/driver/screen/screen_pointer.c	1132
94.4.37	kernel/driver/screen/screen_public.c	1132
94.4.38	kernel/driver/screen/screen_putc.c	1133
94.4.39	kernel/driver/screen/screen_scroll.c	1134
94.4.40	kernel/driver/screen/screen_select.c	1136
94.4.41	kernel/driver/screen/screen_update.c	1137
94.4.42	kernel/driver/tty.h	1139
94.4.43	kernel/driver/tty/tty_console.c	1140
94.4.44	kernel/driver/tty/tty_init.c	1141
94.4.45	kernel/driver/tty/tty_public.c	1143

94.4.46	kernel/driver/tty/tty_read.c	1144
94.4.47	kernel/driver/tty/tty_reference.c	1145
94.4.48	kernel/driver/tty/tty_write.c	1146
94.5	os32: «kernel/fs.h»	1147
94.5.1	kernel/fs/fd_dup.c	1159
94.5.2	kernel/fs/fd_reference.c	1161
94.5.3	kernel/fs/file_pipe_make.c	1162
94.5.4	kernel/fs/file_reference.c	1163
94.5.5	kernel/fs/file_stdio_dev_make.c	1164
94.5.6	kernel/fs/fs_init.c	1166
94.5.7	kernel/fs/fs_public.c	1167
94.5.8	kernel/fs/inode_alloc.c	1167
94.5.9	kernel/fs/inode_check.c	1172
94.5.10	kernel/fs/inode_dir_empty.c	1175
94.5.11	kernel/fs/inode_file_read.c	1177
94.5.12	kernel/fs/inode_file_write.c	1181
94.5.13	kernel/fs/inode_free.c	1184
94.5.14	kernel/fs/inode_fzones_read.c	1185
94.5.15	kernel/fs/inode_fzones_write.c	1187
94.5.16	kernel/fs/inode_get.c	1189
94.5.17	kernel/fs/inode_pipe_make.c	1195
94.5.18	kernel/fs/inode_pipe_read.c	1197
94.5.19	kernel/fs/inode_pipe_write.c	1200
94.5.20	kernel/fs/inode_print.c	1203
94.5.21	kernel/fs/inode_put.c	1206

94.5.22	kernel/fs/inode_reference.c	1208
94.5.23	kernel/fs/inode_save.c	1211
94.5.24	kernel/fs/inode_stdio_dev_make.c	1213
94.5.25	kernel/fs/inode_truncate.c	1215
94.5.26	kernel/fs/inode_zone.c	1219
94.5.27	kernel/fs/path_device.c	1234
94.5.28	kernel/fs/path_fix.c	1235
94.5.29	kernel/fs/path_full.c	1237
94.5.30	kernel/fs/path_inode.c	1239
94.5.31	kernel/fs/path_inode_link.c	1245
94.5.32	kernel/fs/sb_inode_status.c	1253
94.5.33	kernel/fs/sb_mount.c	1255
94.5.34	kernel/fs/sb_print.c	1260
94.5.35	kernel/fs/sb_reference.c	1261
94.5.36	kernel/fs/sb_save.c	1263
94.5.37	kernel/fs/sb_zone_status.c	1265
94.5.38	kernel/fs/sock_free_port.c	1266
94.5.39	kernel/fs/sock_reference.c	1267
94.5.40	kernel/fs/zone_alloc.c	1268
94.5.41	kernel/fs/zone_free.c	1271
94.5.42	kernel/fs/zone_print.c	1273
94.5.43	kernel/fs/zone_read.c	1274
94.5.44	kernel/fs/zone_write.c	1275
94.6	os32: «kernel/ibm_i386.h»	1277
94.6.1	kernel/ibm_i386/_in_16.s	1283

94.6.2	kernel/ibm_i386/_in_32.s	1284
94.6.3	kernel/ibm_i386/_in_8.s	1285
94.6.4	kernel/ibm_i386/_out_16.s	1286
94.6.5	kernel/ibm_i386/_out_32.s	1287
94.6.6	kernel/ibm_i386/_out_8.s	1288
94.6.7	kernel/ibm_i386/cli.s	1288
94.6.8	kernel/ibm_i386/gdt.c	1289
94.6.9	kernel/ibm_i386/gdt_load.s	1290
94.6.10	kernel/ibm_i386/gdt_print.c	1291
94.6.11	kernel/ibm_i386/gdt_public.c	1292
94.6.12	kernel/ibm_i386/gdt_segment.c	1292
94.6.13	kernel/ibm_i386/idt.c	1294
94.6.14	kernel/ibm_i386/idt_descriptor.c	1296
94.6.15	kernel/ibm_i386/idt_irq_remap.c	1298
94.6.16	kernel/ibm_i386/idt_load.s	1300
94.6.17	kernel/ibm_i386/idt_print.c	1300
94.6.18	kernel/ibm_i386/idt_public.c	1301
94.6.19	kernel/ibm_i386/irq_off.c	1301
94.6.20	kernel/ibm_i386/irq_on.c	1302
94.6.21	kernel/ibm_i386/isr.s	1303
94.6.22	kernel/ibm_i386/isr_exception_name.c	1327
94.6.23	kernel/ibm_i386/isr_exception_unrecoverable.c	1328
94.6.24	kernel/ibm_i386/isr_irq_clear.c	1329
94.6.25	kernel/ibm_i386/isr_irq_clear_pic1.c	1330
94.6.26	kernel/ibm_i386/isr_irq_clear_pic2.c	1331

94.6.27	kernel/ibm_i386/sti.s	1331
94.7	os32: «kernel/lib_k.h»	1332
94.7.1	kernel/lib_k/k_exit.s	1333
94.7.2	kernel/lib_k/k_gets.c	1333
94.7.3	kernel/lib_k/k_perror.c	1334
94.7.4	kernel/lib_k/k_printf.c	1335
94.7.5	kernel/lib_k/k_sleep.c	1336
94.7.6	kernel/lib_k/k_stime.c	1337
94.7.7	kernel/lib_k/k_usleep.c	1337
94.7.8	kernel/lib_k/k_vprintf.c	1339
94.7.9	kernel/lib_k/k_vsprintf.c	1340
94.8	os32: «kernel/lib_s.h»	1341
94.8.1	kernel/lib_s/s__exit.c	1346
94.8.2	kernel/lib_s/s_accept.c	1351
94.8.3	kernel/lib_s/s_bind.c	1355
94.8.4	kernel/lib_s/s_brk.c	1359
94.8.5	kernel/lib_s/s_chdir.c	1367
94.8.6	kernel/lib_s/s_chmod.c	1369
94.8.7	kernel/lib_s/s_chown.c	1370
94.8.8	kernel/lib_s/s_clock.c	1372
94.8.9	kernel/lib_s/s_close.c	1372
94.8.10	kernel/lib_s/s_connect.c	1375
94.8.11	kernel/lib_s/s_dup.c	1381
94.8.12	kernel/lib_s/s_dup2.c	1381

94.8.13	kernel/lib_s/s_fchmod.c	1383
94.8.14	kernel/lib_s/s_fchown.c	1384
94.8.15	kernel/lib_s/s_fcntl.c	1386
94.8.16	kernel/lib_s/s_fork.c	1388
94.8.17	kernel/lib_s/s_fstat.c	1398
94.8.18	kernel/lib_s/s_ipconfig.c	1400
94.8.19	kernel/lib_s/s_kill.c	1402
94.8.20	kernel/lib_s/s_link.c	1406
94.8.21	kernel/lib_s/s_listen.c	1408
94.8.22	kernel/lib_s/s_longjmp.c	1410
94.8.23	kernel/lib_s/s_lseek.c	1412
94.8.24	kernel/lib_s/s_mkdir.c	1414
94.8.25	kernel/lib_s/s_mknod.c	1418
94.8.26	kernel/lib_s/s_mount.c	1421
94.8.27	kernel/lib_s/s_open.c	1423
94.8.28	kernel/lib_s/s_pipe.c	1432
94.8.29	kernel/lib_s/s_read.c	1435
94.8.30	kernel/lib_s/s_recvfrom.c	1441
94.8.31	kernel/lib_s/s_routeadd.c	1456
94.8.32	kernel/lib_s/s_routedel.c	1458
94.8.33	kernel/lib_s/s_sbrk.c	1460
94.8.34	kernel/lib_s/s_send.c	1462
94.8.35	kernel/lib_s/s_setegid.c	1469
94.8.36	kernel/lib_s/s seteuid.c	1470
94.8.37	kernel/lib_s/s_setgid.c	1471

94.8.38	kernel/lib_s/s_setjmp.c	1472
94.8.39	kernel/lib_s/s_setuid.c	1474
94.8.40	kernel/lib_s/s_signal.c	1475
94.8.41	kernel/lib_s/s_socket.c	1477
94.8.42	kernel/lib_s/s_stat.c	1480
94.8.43	kernel/lib_s/s_stime.c	1483
94.8.44	kernel/lib_s/s_tcgetattr.c	1484
94.8.45	kernel/lib_s/s_tcsetattr.c	1486
94.8.46	kernel/lib_s/s_time.c	1488
94.8.47	kernel/lib_s/s_umount.c	1489
94.8.48	kernel/lib_s/s_unlink.c	1493
94.8.49	kernel/lib_s/s_wait.c	1498
94.8.50	kernel/lib_s/s_write.c	1500
94.9	os32: «kernel/main.h»	1505
94.9.1	kernel/main/build.h	1506
94.9.2	kernel/main/crt0.s	1506
94.9.3	kernel/main/kmain.c	1508
94.9.4	kernel/main/menu.c	1521
94.9.5	kernel/main/run.c	1522
94.9.6	kernel/main/stack.s	1523
94.10	os32: «kernel/memory.h»	1523
94.10.1	kernel/memory/mb_alloc.c	1525
94.10.2	kernel/memory/mb_alloc_size.c	1528
94.10.3	kernel/memory/mb_clean.c	1531

94.10.4	kernel/memory/mb_free.c	1531
94.10.5	kernel/memory/mb_print.c	1534
94.10.6	kernel/memory/mb_public.c	1536
94.10.7	kernel/memory/mb_reduce.c	1536
94.10.8	kernel/memory/mb_reference.c	1538
94.10.9	kernel/memory/mb_size.c	1538
94.11	os32: «kernel/multiboot.h»	1539
94.11.1	kernel/multiboot/mboot_cmdline_opt.c	1540
94.11.2	kernel/multiboot/mboot_public.c	1543
94.11.3	kernel/multiboot/mboot_save.c	1543
94.12	os32: «kernel/net.h»	1544
94.12.1	kernel/net/arp.h	1552
94.12.2	kernel/net/arp/arp_clean.c	1553
94.12.3	kernel/net/arp/arp_index.c	1554
94.12.4	kernel/net/arp/arp_init.c	1556
94.12.5	kernel/net/arp/arp_print.c	1556
94.12.6	kernel/net/arp/arp_public.c	1557
94.12.7	kernel/net/arp/arp_reference.c	1557
94.12.8	kernel/net/arp/arp_request.c	1558
94.12.9	kernel/net/arp/arp_rx.c	1560
94.12.10	kernel/net/icmp.h	1565
94.12.11	kernel/net/icmp/icmp_rx.c	1566
94.12.12	kernel/net/icmp/icmp_tx.c	1572
94.12.13	kernel/net/icmp/icmp_tx_echo.c	1573

94.12.14	kernel/net/icmp/icmp_tx_unreachable.c	1574
94.12.15	kernel/net/ip.h	1575
94.12.16	kernel/net/ip/ip_checksum.c	1578
94.12.17	kernel/net/ip/ip_header.c	1580
94.12.18	kernel/net/ip/ip_mask.c	1581
94.12.19	kernel/net/ip/ip_public.c	1582
94.12.20	kernel/net/ip/ip_reference.c	1582
94.12.21	kernel/net/ip/ip_rx.c	1583
94.12.22	kernel/net/ip/ip_tx.c	1590
94.12.23	kernel/net/net_buffer_eth.c	1594
94.12.24	kernel/net/net_buffer_lo.c	1595
94.12.25	kernel/net/net_eth_ip_tx.c	1597
94.12.26	kernel/net/net_eth_tx.c	1601
94.12.27	kernel/net/net_index.c	1602
94.12.28	kernel/net/net_index_eth.c	1602
94.12.29	kernel/net/net_init.c	1605
94.12.30	kernel/net/net_print.c	1611
94.12.31	kernel/net/net_public.c	1612
94.12.32	kernel/net/net_rx.c	1612
94.12.33	kernel/net/route.h	1616
94.12.34	kernel/net/route/route_init.c	1617
94.12.35	kernel/net/route/route_print.c	1618
94.12.36	kernel/net/route/route_public.c	1619
94.12.37	kernel/net/route/route_remote_to_local.c	1619
94.12.38	kernel/net/route/route_remote_to_router.c	...	1621

94.12.39	kernel/net/route/route_sort.c	1622
94.12.40	kernel/net/tcp.h	1627
94.12.41	kernel/net/tcp/tcp.c	1628
94.12.42	kernel/net/tcp/tcp_close.c	1653
94.12.43	kernel/net/tcp/tcp_connect.c	1656
94.12.44	kernel/net/tcp/tcp_rx_ack.c	1658
94.12.45	kernel/net/tcp/tcp_rx_data.c	1662
94.12.46	kernel/net/tcp/tcp_show.c	1664
94.12.47	kernel/net/tcp/tcp_status.c	1666
94.12.48	kernel/net/tcp/tcp_test.c	1668
94.12.49	kernel/net/tcp/tcp_tx_ack.c	1669
94.12.50	kernel/net/tcp/tcp_tx_raw.c	1671
94.12.51	kernel/net/tcp/tcp_tx_rst.c	1674
94.12.52	kernel/net/tcp/tcp_tx_sock.c	1677
94.12.53	kernel/net/udp.h	1682
94.12.54	kernel/net/udp/udp_tx.c	1683
94.13	os32: «kernel/part.h»	1685
94.14	os32: «kernel/proc.h»	1686
94.14.1	kernel/proc/proc_available.c	1692
94.14.2	kernel/proc/proc_dump_memory.c	1693
94.14.3	kernel/proc/proc_init.c	1695
94.14.4	kernel/proc/proc_print.c	1701
94.14.5	kernel/proc/proc_public.c	1705
94.14.6	kernel/proc/proc_reference.c	1705

94.14.7	kernel/proc/proc_sch_net.c	1706
94.14.8	kernel/proc/proc_sch_signals.c	1709
94.14.9	kernel/proc/proc_sch_terminals.c	1710
94.14.10	kernel/proc/proc_sch_timers.c	1721
94.14.11	kernel/proc/proc_scheduler.c	1722
94.14.12	kernel/proc/proc_sig_chld.c	1728
94.14.13	kernel/proc/proc_sig_cont.c	1730
94.14.14	kernel/proc/proc_sig_core.c	1731
94.14.15	kernel/proc/proc_sig_handler.c	1733
94.14.16	kernel/proc/proc_sig_ignore.c	1740
94.14.17	kernel/proc/proc_sig_off.c	1740
94.14.18	kernel/proc/proc_sig_on.c	1741
94.14.19	kernel/proc/proc_sig_status.c	1741
94.14.20	kernel/proc/proc_sig_stop.c	1742
94.14.21	kernel/proc/proc_sig_term.c	1742
94.14.22	kernel/proc/proc_sys_exec.c	1744
94.14.23	kernel/proc/proc_timer_init.c	1767
94.14.24	kernel/proc/proc_wakeup_pipe_read.c	1768
94.14.25	kernel/proc/proc_wakeup_pipe_write.c	1769
94.14.26	kernel/proc/proc_wakeup_terminal.c	1769
94.14.27	kernel/proc/ptr.c	1771
94.14.28	kernel/proc/sysroutine.c	1771

94.1 os32: directory principale

<<

94.1.1 applic.sep.ld

<<

Si veda la sezione [84.1.3](#).

```
10001 /******  
10002  * SEPARATED text from data  
10003  *****/  
10004  
10005 ENTRY (startup)  
10006 SECTIONS {  
10007     . = 0x0;  
10008     _text_start = .;  
10009     .text : {  
10010         *(.text)  
10011         . = ALIGN (0x4);  
10012     }  
10013     _text_end = .;  
10014     . = 0x0;  
10015     _data_start = .;  
10016     .rodata : {  
10017         *(.rodata)  
10018         . = ALIGN (0x4);  
10019     }  
10020     .data : {  
10021         *(.data)  
10022         . = ALIGN (0x4);  
10023     }  
10024     _data_end = .;  
10025     _bss_start = .;  
10026     .bss : {  
10027         *(COMMON)  
10028         *(.bss)  
10029         . = ALIGN (0x4);  
10030     }  
10031     _bss_end = .;  
10032 }
```

94.1.2 bochs



Si veda la sezione [85.4](#).

```
20001 #!/bin/sh
20002
20003 if [ "$UID" = 0 ]
20004 then
20005     #   172.21.11.18                               172.21.11.16
20006     #   >-----point to point -----> >-----os32
20007     #   tap0 (linux)                               net1
20008     #
20009     # Dal lato Linux:
20010     #   ifconfig tap0 172.21.11.18 pointopoint \
20011     #           172.21.11.16 netmask 255.255.255.255
20012     #   route add -host 172.21.11.16 gw 172.21.11.18
20013     #
20014     # Dalla macchina 172.21.254.254:
20015     #   route add -host 172.21.11.16 gw 172.21.11.18
20016     #
20017     bochs -q \
20018         "boot: disk" \
20019         "ata0-master: type=disk, path=disk.hda" \
20020         "keyboard_mapping: enabled=1, \
20021         map=/usr/share/bochs/keymaps/x11-pc-it.map" \
20022         "keyboard_type: mf" \
20023         "vga: none" \
20024         "ne2k: mac=b0:c4:20:00:00:00, ioaddr=0x300, \
20025         irq=9, ethmod=tuntap, ethdev=/dev/net/tun, \
20026         script=./tap0" \
20027         "i440fxsupport: enabled=1, slot1=pcivga, \
20028         slot2=ne2k" \
20029         "romimage: \
20030         file=\"/usr/share/bochs/BIOS-bochs-legacy\"" \
20031         "megs:128"
20032 else
20033     bochs -q \
20034         "boot: disk" \
```

```

20035     "ata0-master: type=disk, path=disk.hda" \
20036     "keyboard_mapping: enabled=1, \
20037         map=/usr/share/bochs/keymaps/x11-pc-it.map" \
20038     "keyboard_type: mf" \
20039     "vga: none" \
20040     "ne2k: mac=b0:c4:20:00:00:00, ioaddr=0x300, \
20041         irq=9, ethmod=null" \
20042     "i440fxsupport: enabled=1, slot1=pcivga, \
20043         slot2=ne2k" \
20044     "romimage: \
20045         file=\"/usr/share/bochs/BIOS-bochs-legacy\" \" \" \
20046     "megs:128"
20047 fi

```

94.1.3 elf-to-os32



Si veda la sezione [84.1.3](#).

```

30001 #!/bin/sh
30002 #
30003 #
30004 #
30005 g_sz ()
30006 {
30007     sed "s/^[[:space:]]*[0-9]*[[:space:]]* *//" \
30008     | sed "s/\.^[[:space:]]*[[:space:]]* *//" \
30009     | sed "s/\([0-9a-z]*\) [[:space:]].*$/\1/"
30010 }
30011 #
30012 g_vma ()
30013 {
30014     sed "s/^[[:space:]]*[0-9]*[[:space:]]* *//" \
30015     | sed "s/\.^[[:space:]]*[[:space:]]* *//" \
30016     | sed "s/[0-9a-f]*[[:space:]]* *//" \
30017     | sed "s/\([0-9a-z]*\) [[:space:]].*$/\1/"
30018 }
30019 #

```

```
30020 g_st ()
30021 {
30022     sed "s/^[[:space:]]*[0-9]*[[:space:]]*// " \
30023     | sed "s/\.^[[:space:]]*[[:space:]]*// " \
30024     | sed "s/[0-9a-f]*[[:space:]]*// " \
30025     | sed "s/[0-9a-f]*[[:space:]]*// " \
30026     | sed "s/[0-9a-f]*[[:space:]]*// " \
30027     | sed "s/\([0-9a-z]*\) [[:space:]].*$/\1/"
30028 }
30029 #
30030 FILE_ELF="$1"
30031 FILE_OS32="$2"
30032 #
30033 if [ -e "$FILE_ELF" ]
30034 then
30035     true
30036 else
30037     exit
30038 fi
30039 #
30040 T_ST=`objdump -h $FILE_ELF | grep -F ".text" | g_st `
30041 T_VM=`objdump -h $FILE_ELF | grep -F ".text" | g_vma `
30042 T_SZ=`objdump -h $FILE_ELF | grep -F ".text" | g_sz `
30043 #
30044 R_ST=`objdump -h $FILE_ELF | grep -F ".rodata" | g_st `
30045 R_VM=`objdump -h $FILE_ELF | grep -F ".rodata" | g_vma `
30046 R_SZ=`objdump -h $FILE_ELF | grep -F ".rodata" | g_sz `
30047 #
30048 D_ST=`objdump -h $FILE_ELF | grep -F ".data" | g_st `
30049 D_VM=`objdump -h $FILE_ELF | grep -F ".data" | g_vma `
30050 D_SZ=`objdump -h $FILE_ELF | grep -F ".data" | g_sz `
30051 #
30052 # Convert to decimal
30053 #
30054 T_ST=`printf "%i" 0x$T_ST `
30055 T_VM=`printf "%i" 0x$T_VM `
30056 T_SZ=`printf "%i" 0x$T_SZ `
```

```
30057 #
30058 R_ST=`printf "%i" 0x$R_ST`
30059 R_VM=`printf "%i" 0x$R_VM`
30060 R_SZ=`printf "%i" 0x$R_SZ`
30061 #
30062 D_ST=`printf "%i" 0x$D_ST`
30063 D_VM=`printf "%i" 0x$D_VM`
30064 D_SZ=`printf "%i" 0x$D_SZ`
30065
30066
30067 if [ "$R_SZ" = "$D_VM" ]
30068 then
30069     dd if=$FILE_ELF of=$FILE_OS32.text \
30070         bs=1 skip=$T_ST count=$T_SZ 2> "/dev/null"
30071     dd if=$FILE_ELF of=$FILE_OS32.rodata \
30072         bs=1 skip=$R_ST count=$R_SZ 2> "/dev/null"
30073     dd if=$FILE_ELF of=$FILE_OS32.data \
30074         bs=1 skip=$D_ST count=$D_SZ 2> "/dev/null"
30075     cat $FILE_OS32.text $FILE_OS32.rodata \
30076         $FILE_OS32.data > $FILE_OS32
30077     #
30078     #rm $FILE_OS32.text
30079     #rm $FILE_OS32.rodata
30080     #rm $FILE_OS32.data
30081     #
30082     chmod a+x $FILE_OS32
30083 elif [ "$R_SZ" -lt "$D_VM" ]
30084 then
30085     dd if=$FILE_ELF of=$FILE_OS32.text \
30086         bs=1 skip=$T_ST count=$T_SZ 2> "/dev/null"
30087     dd if=$FILE_ELF of=$FILE_OS32.rodata \
30088         bs=1 skip=$R_ST count=$R_SZ 2> "/dev/null"
30089     dd if=/dev/zero of=$FILE_OS32.rodata-space \
30090         bs=1 count=$(( $D_VM-$R_SZ )) 2> "/dev/null"
30091     dd if=$FILE_ELF of=$FILE_OS32.data \
30092         bs=1 skip=$D_ST count=$D_SZ 2> "/dev/null"
30093
```



```
30094     cat $FILE_OS32.text \  
30095         $FILE_OS32.rodata \  
30096         $FILE_OS32.rodata-space \  
30097         $FILE_OS32.data > $FILE_OS32  
30098     #  
30099     #rm $FILE_OS32.text  
30100     #rm $FILE_OS32.rodata  
30101     #rm $FILE_OS32.rodata-space  
30102     #rm $FILE_OS32.data  
30103     #  
30104     chmod a+x $FILE_OS32  
30105 else  
30106     echo "[\$0] ERROR: $FILE_ELF has DATA section"  
30107     echo "[\$0]         not contiguous:"  
30108     echo "[\$0]         RODATA end at $R_SZ, and DATA"  
30109     echo "[\$0]         should start at $D_VM!"  
30110 fi
```

94.1.4 fdisk

Si veda la sezione [85.1](#).

```
40001 #!/bin/sh  
40002 #  
40003 #  
40004 #  
40005 . ./file_image_functions  
40006 #  
40007 if [ -z "$1" ]  
40008 then  
40009     echo "$0 DISK_IMAGE_FILE"  
40010 else  
40011     file_image_fdisk "$1"  
40012 fi  
40013 #
```



94.1.5 file_image_functions



Si veda la sezione [85.1](#).

```
50001 #
50002 # file_image_fdisk IMAGE_FILE
50003 #
50004 file_image_fdisk () {
50005     #
50006     local FNAME="$1"
50007     local FSIZE=0
50008     local CYLINDERS=0
50009     local HEADS=0
50010     local SECTORS=63
50011     #
50012     if [ -z "$FNAME" ]
50013     then
50014         echo "[\$0] No file name."
50015         return 1
50016     elif [ ! -r "$FNAME" ]
50017     then
50018         echo "[\$0] Cannot read file \"$FNAME\"."
50019         return 1
50020     fi
50021     #
50022     # Get size
50023     #
50024     FSIZE=`du -k "$FNAME" | sed "s/[[:space:]].*$//"`
50025     #
50026     # Set geometry
50027     #
50028     if [ "$FSIZE" -le "516096" ]
50029     then
50030         HEADS="16"
50031     else
50032         HEADS="255"
50033     fi
50034     #
```

```
50035     CYLINDERS=$((($FSIZE*2/$SECTORS/$HEADS))
50036     #
50037     # Run fdisk.
50038     #
50039     fdisk -C $CYLINDERS -H $HEADS -S $SECTORS $FNAME
50040 }
50041 #
50042 # file_image_partition_start IMAGE_FILE PART_NUMBER
50043 #
50044 file_image_partition_start () {
50045     #
50046     local FNAME="$1"
50047     local PART_NUMBER="$2"
50048     local PART_START=0
50049     #
50050     # Get partition start
50051     #
50052     PART_START=`sfdisk -d $FNAME \
50053         | grep -F "$FNAME$PART_NUMBER" \
50054         | sed "s/^. *start=[[[:space:]]]*//" | sed "s/,.*$//"`
50055     #
50056     echo "$PART_START"
50057 }
50058 #
50059 # file_image_partition_size IMAGE_FILE PART_NUMBER
50060 #
50061 file_image_partition_size () {
50062     #
50063     local FNAME="$1"
50064     local PART_NUMBER="$2"
50065     local PART_SIZE=0
50066     #
50067     # Get partition start
50068     #
50069     PART_SIZE=`sfdisk -d $FNAME \
50070         | grep -F "$FNAME$PART_NUMBER" \
50071         | sed "s/^. *size=[[[:space:]]]*//" | sed "s/,.*$//"`
```

```
50072 #
50073 echo "$PART_SIZE"
50074 }
50075 #
50076 # file_image_partition_id IMAGE_FILE PART_NUMBER
50077 #
50078 file_image_partition_id () {
50079 #
50080 local FNAME="$1"
50081 local PART_NUMBER="$2"
50082 local PART_ID=0
50083 #
50084 # Get partition start
50085 #
50086 PART_ID=`sfdisk -d $FNAME \
50087 | grep -F "$FNAME$PART_NUMBER" \
50088 | sed "s/^. *Id=[[[:space:]]*// " | sed "s/, .*$//"`
50089 #
50090 echo "$PART_ID"
50091 }
50092 #
50093 # file_image_partition_format IMAGE_FILE PART_NUMBER \
50094 # [dos|minix]
50095 #
50096 file_image_partition_format () {
50097 #
50098 local FNAME="$1"
50099 local PART_NUMBER="$2"
50100 local PART_FORMAT="$3"
50101 local PART_START=`file_image_partition_start \
50102 $FNAME $PART_NUMBER`
50103 local PART_SIZE=`file_image_partition_size $FNAME \
50104 $PART_NUMBER`
50105 local PART_ID=`file_image_partition_id $FNAME \
50106 $PART_NUMBER`
50107 #
50108 #
```

```
50109 #
50110 if [ "$PART_SIZE" -eq "0" ]
50111 then
50112     exit
50113 fi
50114 #
50115 # Get partition into a file.
50116 #
50117 dd if="$FNAME" \
50118     of="$FNAME.part$PART_NUMBER.tmp" \
50119     bs=512 \
50120     skip="$PART_START" \
50121     count="$PART_SIZE"
50122 #
50123 # Format.
50124 #
50125 if [ "$PART_FORMAT" = "dos" ] \
50126     || [ "$PART_FORMAT" = "msdos" ]
50127 then
50128     mkfs.msdos -v "$FNAME.part$PART_NUMBER.tmp"
50129 else
50130     mkfs.minix -n 14 "$FNAME.part$PART_NUMBER.tmp"
50131 fi
50132 #
50133 # Put formatted partition into the file.
50134 #
50135 dd if="$FNAME.part$PART_NUMBER.tmp" \
50136     of="$FNAME" \
50137     bs=512 \
50138     seek="$PART_START" \
50139     count="$PART_SIZE" \
50140     conv=notrunc
50141 #
50142 # Remove temporary file.
50143 #
50144 rm "$FNAME.part$PART_NUMBER.tmp"
50145 #
```

```
50146 }
50147 #
50148 # file_image_partition_mount IMAGE_FILE PART_NUMBER
50149 #
50150 file_image_partition_mount () {
50151     #
50152     local FNAME="$1"
50153     local PART_NUMBER="$2"
50154     local PART_START=`file_image_partition_start \
50155         $FNAME $PART_NUMBER`
50156     local PART_SIZE=`file_image_partition_size \
50157         $FNAME $PART_NUMBER`
50158     local PART_ID=`file_image_partition_id \
50159         $FNAME $PART_NUMBER`
50160     local MOUNT_POINT
50161     #
50162     #
50163     #
50164     if [ "$PART_SIZE" -eq "0" ]
50165     then
50166         exit
50167     fi
50168     #
50169     # Find mount point.
50170     #
50171     MOUNT_POINT=`basename $FNAME`
50172     MOUNT_POINT="/mnt/${MOUNT_POINT}.$PART_NUMBER"
50173     #
50174     #
50175     #
50176     sync
50177     #
50178     umount "$MOUNT_POINT" 2> "/dev/null"
50179     umount "$MOUNT_POINT" 2> "/dev/null"
50180     umount "$MOUNT_POINT" 2> "/dev/null"
50181     #
50182     mkdir -p "$MOUNT_POINT" 2> "/dev/null"
```

```
50183 #
50184 # Get partition into a file.
50185 #
50186 dd if="$FNAME" \
50187     of="$FNAME.part$PART_NUMBER.tmp" \
50188     bs=512 \
50189     skip="$PART_START" \
50190     count="$PART_SIZE"
50191 #
50192 # Check partition.
50193 #
50194 fsck -f -v -r "$FNAME.part$PART_NUMBER.tmp"
50195 #
50196 # Put formatted partition into the file.
50197 #
50198 dd if="$FNAME.part$PART_NUMBER.tmp" \
50199     of="$FNAME" \
50200     bs=512 \
50201     seek="$PART_START" \
50202     count="$PART_SIZE" \
50203     conv=notrunc
50204 #
50205 # Remove temporary file.
50206 #
50207 rm "$FNAME.part$PART_NUMBER.tmp"
50208 #
50209 # Mount the partition.
50210 #
50211 mount -o loop,offset=$((($PART_START*512)) \
50212     -t auto "$FNAME" "$MOUNT_POINT"
50213 #
50214 }
50215 #
50216 # file_image_partition_syslinux IMAGE_FILE PART_NUMBER
50217 #
50218 file_image_partition_syslinux () {
50219 #
```

```
50220 local FNAME="$1"
50221 local PART_NUMBER="$2"
50222 local PART_START=`file_image_partition_start \
50223             $FNAME $PART_NUMBER`
50224 local PART_SIZE=`file_image_partition_size \
50225             $FNAME $PART_NUMBER`
50226 local PART_ID=`file_image_partition_id \
50227             $FNAME $PART_NUMBER`
50228 #
50229 #
50230 #
50231 if [ "$PART_SIZE" -eq "0" ]
50232 then
50233     exit
50234 fi
50235 #
50236 # Get partition into a file.
50237 #
50238 dd if="$FNAME" \
50239    of="$FNAME.part$PART_NUMBER.tmp" \
50240    bs=512 \
50241    skip="$PART_START" \
50242    count="$PART_SIZE"
50243 #
50244 # Syslinux
50245 #
50246 syslinux "$FNAME.part$PART_NUMBER.tmp"
50247 #
50248 # Put altered partition into the file.
50249 #
50250 dd if="$FNAME.part$PART_NUMBER.tmp" \
50251    of="$FNAME" \
50252    bs=512 \
50253    seek="$PART_START" \
50254    count="$PART_SIZE" \
50255    conv=notrunc
50256 #
```



```
50257 # Remove temporary file.
50258 #
50259 rm "$FNAME.part$PART_NUMBER.tmp"
50260 #
50261 # Fix MBR
50262 #
50263 install-mbr $FNAME
50264 }
50265
```

94.1.6 format



Si veda la sezione [85.1](#).

```
60001 #!/bin/sh
60002 #
60003 #
60004 #
60005 . ./file_image_functions
60006 #
60007 if [ -z "$1" ] \
60008     || [ -z "$2" ] \
60009     || [ "$2" -lt "1" ] \
60010     || [ "$2" -gt "4" ]
60011 then
60012     echo "Usage:"
60013     echo ""
60014     echo "$0 DISK_IMAGE_FILE PART_NUMBER dos|minix"
60015     echo ""
60016     echo "The partition number must be between"
60017     echo " 1 and 4. No extended partitions are"
60018     echo "handled!"
60019 else
60020     file_image_partition_format "$1" "$2" "$3"
60021 fi
60022 #
```

94.1.7 kernel.ld



Si veda la sezione [84.2.2](#).

```
70001 /******
70002  * The code will start at address 0x100000, that is at
70003  * 1 Mibyte, because it is the place where GRUB will
70004  * place it.
70005  *
70006  * The kernel is divided into 'TEXT' (code), 'DATA' and
70007  * 'BSS'.
70008  * Between the TEXT and the DATA there is a gap to
70009  * align the data at 4 Kibyte boundary (0x1000), to
70010  * allow memory management for it.
70011  *
70012  * The stack will be placed at the beginning of the
70013  * BSS.
70014  *
70015  * The kernel starts with file 'kernel/main/crt0.s',
70016  * at the label 'startup'.
70017  *****/
70018 ENTRY (kstartup)
70019 SECTIONS {
70020     . = 0x00100000;
70021     _k_start = .;
70022     _k_text_start = .;
70023     .text : {
70024         *(.text)
70025     }
70026     _k_text_end = .;
70027     . = ALIGN (0x1000);
70028     _k_data_start = .;
70029     .rodata : {
70030         *(.rodata)
70031     }
70032     . = ALIGN (0x4);
70033     .data : {
70034         *(.data)
```

```

70035     }
70036     _k_data_end = .;
70037     . = ALIGN (0x4);
70038     _k_bss_start = .;
70039     .bss : {
70040         *(COMMON)
70041         *(.bss)
70042     }
70043     _k_bss_end = .;
70044     _k_end = .;
70045 }

```

94.1.8 makeit.sep

Si veda la sezione [91.3](#).



```

80001 #!/bin/sh
80002 #
80003 # makeit... separated: text and data have separate
80004 # segments.
80005 #
80006 OPTION="$1"
80007 OS32PATH=""
80008 #
80009 #
80010 #
80011 edition () {
80012     local EDITION="kernel/main/build.h"
80013     echo -n                                     > $EDITION
80014     echo -n "#define BUILD_DATE \"\"           >> $EDITION
80015     echo -n `date "+Y%m%d%H%M" `              >> $EDITION
80016     echo  "\"\"  >> $EDITION
80017 }
80018 #
80019 #
80020 #
80021 makefile () {

```

```
80022 #
80023 local MAKEFILE="Makefile"
80024 local TAB=`printf "\t" `
80025 #
80026 local SOURCE_C=""
80027 local C=""
80028 local SOURCE_S=""
80029 local S=""
80030 #
80031 local c
80032 local s
80033 #
80034 # Find C source files.
80035 #
80036 for c in *.c
80037 do
80038     if [ -f $c ]
80039     then
80040         C=`basename $c .c `
80041         SOURCE_C="$SOURCE_C $C"
80042     fi
80043 done
80044 #
80045 # Find ASM source files.
80046 #
80047 for s in *.s
80048 do
80049     if [ -f $s ]
80050     then
80051         S=`basename $s .s `
80052         SOURCE_S="$SOURCE_S $S"
80053     fi
80054 done
80055 #
80056 # Prepare the Makefile. Option '-g' is for debugging
80057 # symbols.
80058 #
```

```

80059     echo -n                                     > $MAKEFILE
80060     echo "# This file was made "              >> $MAKEFILE
80061     echo "# automatically"                   >> $MAKEFILE
80062     echo "# by the script \'makeit\', based"  >> $MAKEFILE
80063     echo "# on the directory content."       >> $MAKEFILE
80064     echo "# Please use \'makeit\' to "       >> $MAKEFILE
80065     echo "# compile and"                     >> $MAKEFILE
80066     echo "# \'makeit clean\' to clean "      >> $MAKEFILE
80067     echo "# directories."                    >> $MAKEFILE
80068     echo "#"                                  >> $MAKEFILE
80069     echo "#"                                  >> $MAKEFILE
80070     echo "c = $SOURCE_C"                     >> $MAKEFILE
80071     echo "#"                                  >> $MAKEFILE
80072     echo "s = $SOURCE_S"                     >> $MAKEFILE
80073     echo "#"                                  >> $MAKEFILE
80074     echo "all: \$(s) \$(c) "                 >> $MAKEFILE
80075     echo "#"                                  >> $MAKEFILE
80076     echo "clean:"                             >> $MAKEFILE
80077     echo "${TAB}@rm *~ *.o *.ELF *.text " \
80078     "*.rodata *.rodata-space " \
80079     "*.data \$(c) 2> /dev/null ; pwd"       >> $MAKEFILE
80080     echo "#"                                  >> $MAKEFILE
80081     echo "\$(s) :"                            >> $MAKEFILE
80082     echo "${TAB}@echo \${@.s}"                >> $MAKEFILE
80083     echo "${TAB}@as -o \${@.o} \${@.s}"      >> $MAKEFILE
80084     echo "#"                                  >> $MAKEFILE
80085     echo "\$(c) :"                            >> $MAKEFILE
80086     echo "${TAB}@echo \${@.c}"                >> $MAKEFILE
80087     echo "${TAB}@gcc -O0 -Wall -Werror " \
80088     "-Wno-unused-but-set-variable" \
80089     "-g" \
80090     "-o \${@.o} -c \${@.c}" \
80091     "-nostdinc -nostdlib " \
80092     "-nostartfiles -nodefaultlibs" \
80093     "-I " \
80094     "-I. " \
80095     "-I$OS32PATH/lib " \

```

```
80096     "-I$OS32PATH/ " \
80097     "-I../include -I../../include " \
80098     "-I../../../include" >> $MAKEFILE
80099     #
80100 }
80101 #
80102 #
80103 #
80104 main () {
80105     #
80106     local CURDIR=`pwd`
80107     local OBJECTS
80108     local d
80109     local c
80110     local s
80111     local o
80112     #
80113     edition
80114     #
80115     # Copia dello scheletro
80116     #
80117     if [ "$OPTION" = "clean" ]
80118     then
80119         #
80120         # La copia non va fatta.
80121         #
80122         true
80123     else
80124         cp -dpRv skel/etc      /mnt/disk.hda.2/
80125         cp -dpRv skel/dev      /mnt/disk.hda.2/
80126         mkdir                  /mnt/disk.hda.2/mnt/
80127         mkdir                  /mnt/disk.hda.2/tmp/
80128         chmod 0777             /mnt/disk.hda.2/tmp/
80129         mkdir                  /mnt/disk.hda.2/usr/
80130         mkdir                  /mnt/disk.hda.2/var/
80131         cp -dpRv skel/root     /mnt/disk.hda.2/
80132         cp -dpRv skel/home     /mnt/disk.hda.2/
```

```
80133     cp -dpRv skel/usr/* /mnt/disk.hda.2/usr/
80134     cp -dpRv skel/var/* /mnt/disk.hda.2/var/
80135 fi
80136 #
80137 for d in `find .`
80138 do
80139     if [ -d "$d" ]
80140     then
80141         #
80142         # Are there C or ASM source files?
80143         #
80144         c=`echo $d/*.*c | sed "s/ .*//"`
80145         s=`echo $d/*.*s | sed "s/ .*//"`
80146         #
80147         if [ -f "$c" ] || [ -f "$s" ]
80148         then
80149             CURDIR=`pwd`
80150             cd $d
80151             #
80152             # Build the new makefile
80153             #
80154             makefile
80155             #
80156             # Clean the directory
80157             #
80158             make clean
80159             #
80160             #
80161             #
80162             if [ "$OPTION" = "clean" ]
80163             then
80164                 #
80165                 # Nothing else to do: the clean was
80166                 # just made.
80167                 #
80168                 true
80169             else
```

```
80170         if ! make
80171         then
80172             cd "$CURDIR"
80173             exit
80174         fi
80175     fi
80176     cd "$CURDIR"
80177 fi
80178 fi
80179 done
80180 #
80181 cd "$CURDIR"
80182 #
80183 #
80184 #
80185 if [ "$OPTION" = "clean" ]
80186 then
80187     true
80188 else
80189     #
80190     echo "Link kernel"
80191     #
80192     OBJECTS=""
80193     #
80194     for o in `find . -name \*.o -print`
80195     do
80196         if [ "$o" = "./kernel/main/crt0.o" ] \
80197         || [ "$o" = "./kernel/main/kmain.o" ] \
80198         || [ "$o" = "./kernel/main/stack.o" ] \
80199         || [ ! -e "$o" ] \
80200         || echo "$o" | grep -F "./applic/" \
80201             > "/dev/null" \
80202         || echo "$o" | grep -F "./ported/" \
80203             > "/dev/null"
80204     then
80205         true
80206     else
```



```
80207     OBJECTS="$OBJECTS $o"
80208     fi
80209 done
80210 #
80211 # The kernel must be ELF, because Grub will not
80212 # recognize it otherwise.
80213 #
80214 ld --script=kernel.ld \
80215     --oformat elf32-i386 \
80216     -o kimage \
80217     ./kernel/main/crt0.o \
80218     $OBJECTS \
80219     ./kernel/main/kmain.o \
80220     ./kernel/main/stack.o
80221 #
80222 cp -f kimage /mnt/disk.hda.1/kimage
80223 sync
80224 #
80225 # Collegamento delle applicazioni di os32.
80226 #
80227 OBJLIB=""
80228 #
80229 for o in `find lib      -name \*.o -print`
80230 do
80231     OBJLIB="$OBJLIB $o"
80232 done
80233 #
80234 echo "Link applic"
80235 #
80236 # Scansione delle applicazioni interne.
80237 #
80238 for o in `find applic  -name \*.o -print`
80239 do
80240     if    [ "$o" = "applic/crt0.o" ] \
80241         || [ ! -e "$o" ] \
80242         || echo "$o" | grep ".crt0.o$" > /dev/null \
80243         || echo "$o" | grep ".crt0.mer.o$" \
```

```
80244         > /dev/null \  
80245     || echo "$o" | grep ".crt0.sep.o$" > /dev/null  
80246 then  
80247     #  
80248     # Il file non esiste oppure si tratta di  
80249     # `...crt0.s`.  
80250     #  
80251     true  
80252 else  
80253     #  
80254     # File oggetto differente da `...crt0.s`.  
80255     #  
80256     EXEC=`echo "$o" | sed "s/\.o$//"`\  
80257     BASENAME=`basename $o .o`\  
80258     if [ -e "applic/$BASENAME.crt0.sep.o" ]  
80259     then  
80260         #  
80261         # Qui c'è un file `...crt0.o` specifico.  
80262         #  
80263         rm $EXEC $EXEC.ELF 2> "/dev/null"  
80264         ld --no-check-sections \  
80265             --oformat elf32-i386 \  
80266             --script=applic.sep.ld \  
80267             -o $EXEC.ELF \  
80268             ./applic/$BASENAME.crt0.sep.o \  
80269             $o \  
80270             $OBJLIB  
80271         #  
80272         ./elf-to-os32 $EXEC.ELF $EXEC  
80273     else  
80274         #  
80275         # Qui si usa il file `crt0.sep.o` generale.  
80276         #  
80277         rm $EXEC $EXEC.ELF 2> "/dev/null"  
80278         ld --script=applic.sep.ld \  
80279             --no-check-sections \  
80280             --oformat elf32-i386 \  

```

```
80281         -o $EXEC.ELF \  
80282         ./applic/crt0.sep.o \  
80283         $o \  
80284         $OBJLIB  
80285         #  
80286         ./elf-to-os32 $EXEC.ELF $EXEC  
80287     fi  
80288     #  
80289     if [ -x "applic/$BASENAME" ]  
80290     then  
80291         if mount | grep /mnt/disk.hda.2 > /dev/null  
80292         then  
80293             mkdir /mnt/disk.hda.2/bin/ 2> /dev/null  
80294             rm /mnt/disk.hda.2/bin/$EXEC 2> "/dev/null"  
80295             cp -f "$EXEC" /mnt/disk.hda.2/bin  
80296         else  
80297             echo "[${0}] Cannot copy the application"  
80298             echo "[${0}]     $BASENAME inside the disk"  
80299             echo "[${0}]     image!"  
80300             break  
80301         fi  
80302     fi  
80303     fi  
80304     done  
80305     sync  
80306     #  
80307     echo "Link ported"  
80308     #  
80309     # Scansione delle applicazioni adattate.  
80310     #  
80311     for a in ported/*  
80312     do  
80313         if [ -d $a ]  
80314         then  
80315             OBJECTS=""  
80316             for o in `find $a -name \*.o -print`  
80317             do
```

```
80318     if      [ "$o" = "$a/crt0.o" ] \  
80319         || [ ! -e "$o" ] \  
80320         || echo "$o" | grep "crt0.o$" > /dev/null \  
80321         || echo "$o" | grep "crt0.mer.o$" \  
80322             > /dev/null \  
80323         || echo "$o" | grep "crt0.sep.o$" \  
80324             > /dev/null  
80325     then  
80326         #  
80327         # Il file non esiste oppure si tratta di  
80328         # `...crt0.s`.  
80329         #  
80330         true  
80331     else  
80332         OBJECTS="$OBJECTS $o"  
80333     fi  
80334 done  
80335 #  
80336 # File oggetto differente da `...crt0.s`.  
80337 #  
80338 BASENAME=`basename $a`  
80339 EXEC="$a/$BASENAME"  
80340 #  
80341 rm $EXEC $EXEC.ELF 2> "/dev/null"  
80342 #  
80343 echo ld --script=applic.sep.ld \  
80344     --no-check-sections \  
80345     --oformat elf32-i386 \  
80346     -o $EXEC.ELF \  
80347     $a/crt0.sep.o \  
80348     $OBJECTS \  
80349     $OBJLIB > link-$BASENAME.link  
80350 #  
80351 echo ./elf-to-os32 $EXEC.ELF $EXEC \  
80352     >> link-$BASENAME.link  
80353 #  
80354 ld --script=applic.sep.ld \  

```

```
80355         --no-check-sections \  
80356         --oformat elf32-i386 \  
80357         -o $EXEC.ELF \  
80358         $a/crt0.sep.o \  
80359         $OBJECTS \  
80360         $OBJLIB  
80361         #  
80362         ./elf-to-os32 $EXEC.ELF $EXEC  
80363         #  
80364         if [ -x "$EXEC" ]  
80365         then  
80366             if mount | grep /mnt/disk.hda.2 > /dev/null  
80367             then  
80368                 mkdir /mnt/disk.hda.2/bin/ 2> /dev/null  
80369                 rm /mnt/disk.hda.2/bin/$EXEC 2> "/dev/null"  
80370                 cp -f "$EXEC" /mnt/disk.hda.2/bin  
80371             else  
80372                 echo "[${0}] Cannot copy the application "  
80373                 echo "[${0}]     $BASENAME inside the disk "  
80374                 echo "[${0}]     image!"  
80375                 break  
80376             fi  
80377         fi  
80378     fi  
80379     done  
80380     sync  
80381     fi  
80382 }  
80383 #  
80384 # Start.  
80385 #  
80386 if [ -d kernel ]    && \  
80387     [ -d lib ]  
80388 then  
80389     OS32PATH=`pwd`  
80390     main  
80391 else
```

```
80392     echo "[\$0] Running from a wrong directory!"
80393 fi
```

94.1.9 qemu

<<

Si veda la sezione [85.4](#).

```
90001 #!/bin/sh
90002 #
90003 #
90004 #
90005 if [ -n "$DISPLAY" ]
90006 then
90007     CURSES=""
90008 else
90009     CURSES="-curses"
90010 fi
90011
90012 if [ "$EUID" = "0" ]
90013 then
90014     # 172.21.11.18                172.21.11.16
90015     # >-----point to point -----> >-----os32
90016     # tap0 (linux)                net1
90017     #
90018     # Dal lato Linux:
90019     # ifconfig tap0 172.21.11.18 pointopoint \
90020     #             172.21.11.16 netmask 255.255.255.255
90021     # route add -host 172.21.11.16 gw 172.21.11.18
90022     #
90023     # Dalla macchina 172.21.254.254:
90024     # route add -host 172.21.11.16 gw 172.21.11.18
90025     #
90026     qemu $CURSES \
90027         -hda disk.hda \
90028         -net nic,macaddr=b0:c4:20:00:00:00,model=ne2k_pci \
90029         -net tap,ifname=tap0,script=./tap0 \
90030         -boot c
```

```
90031 else
90032     echo "[\$0] Qemu avviato senza privilegi: non"
90033     echo "[\$0] funziona la rete!"
90034     echo "[\$0] Premi Invio per continuare"
90035     read
90036
90037     qemu $CURSES \
90038         -hda disk.hda \
90039         -net nic,macaddr=b0:c4:20:00:00:00,model=ne2k_pci \
90040         -net user,net=172.21.0.0/16,host=172.21.254.254,\
90041     restrict=n \
90042         -boot c
90043 fi
90044
```

94.1.10 syslinux

Si veda la sezione [85.1](#).

```
100001 #!/bin/sh
100002 #
100003 #
100004 #
100005 . ./file_image_functions
100006 #
100007 if [ -z "$1" ] || [ -z "$2" ] || [ "$2" -lt "1" ] \
100008     || [ "$2" -gt "4" ]
100009 then
100010     echo "Usage:"
100011     echo ""
100012     echo "$0 DISK_IMAGE_FILE PART_NUMBER"
100013     echo ""
100014     echo "The partition number must be between"
100015     echo " 1 and 4."
100016     echo "No extended partitions are handled!"
100017 else
100018     file_image_partition_syslinux "$1" "$2"
```

100019	fi
100020	#

94.1.11 tap0

<<

Si veda la sezione [85.4](#).

110001	#!/bin/sh
110002	
110003	
110004	ifconfig tap0 172.21.11.18 pointopoint 172.21.11.16 \
110005	netmask 255.255.255.255
110006	route add -host 172.21.11.16 gw 172.21.11.18
110007	
110008	

94.2 os32: «kernel/blk.h»

<<

Si veda la sezione [93.3](#).

120001	#ifndef _KERNEL_BLK_H
120002	#define _KERNEL_BLK_H 1
120003	//-----
120004	#include <sys/types.h>
120005	#include <kernel/driver/ata.h>
120006	//-----
120007	#define BLK_SIZE ATA_SECTOR_SIZE // [1]
120008	//
120009	// [1] <i>This value should be the same as the mass memory</i>
120010	// <i>sector size, or a multiple. But with a multiple,</i>
120011	// <i>all simple read and write will be doubled, and</i>
120012	// <i>some race will lock the system, because read and</i>
120013	// <i>write are too frequent for the poor ATA driver</i>
120014	// <i>that I have. :-)</i>
120015	//
120016	//-----


```

120017 #define BLK_CACHE_SIZE          128      // This is
120018                                     // free!
120019 #define BLK_CACHE_MAX_AGE      BLK_CACHE_SIZE-1
120020 typedef struct
120021 {
120022     unsigned int age;           // 0=last used
120023     // DEV_CACHE_MAX_AGE=older
120024     dev_t device;
120025     unsigned int n;
120026     char block[BLK_SIZE];
120027 } blk_cache_t;
120028
120029 extern blk_cache_t blk_table[BLK_CACHE_SIZE];
120030 //-----
120031 void *blk_ata (dev_t device, int rw, unsigned int n,
120032              void *buffer);
120033 //-----
120034 void blk_cache_init (void);
120035 void blk_cache_check (void);
120036 void *blk_cache_read (dev_t device, unsigned int n);
120037 void *blk_cache_save (dev_t device, unsigned int n,
120038                    void *block);
120039 //-----
120040
120041 #endif

```

94.2.1	kernel/blk/blk_ata.c	1010
94.2.2	kernel/blk/blk_cache_check.c	1012
94.2.3	kernel/blk/blk_cache_init.c	1013
94.2.4	kernel/blk/blk_cache_read.c	1014
94.2.5	kernel/blk/blk_cache_save.c	1015
94.2.6	kernel/blk/blk_public.c	1017

94.2.1 kernel/blk/blk_ata.c



Si veda la sezione [93.3.1](#).

```
130001 #include <sys/os32.h>
130002 #include <kernel/blk.h>
130003 #include <kernel/dev.h>
130004 #include <kernel/driver/ata.h>
130005 #include <kernel/lib_k.h>
130006 #include <kernel/dm.h>
130007 #include <sys/types.h>
130008 #include <ctype.h>
130009 #include <string.h>
130010 //-----
130011 #define DEBUG 0
130012 //-----
130013 void *
130014 blk_ata (dev_t device, int rw, unsigned int n, void *buffer)
130015 {
130016     ata_sector_t *destination = buffer;
130017     ata_sector_t *source = buffer;
130018     int dev_minor = minor (device);
130019     int d = ((dev_minor & 0x00F0) >> 4);
130020     ptrdiff_t ptrdiff;
130021     int drive;
130022     unsigned int sector = n * (BLK_SIZE / ATA_SECTOR_SIZE);
130023     size_t count = (BLK_SIZE / ATA_SECTOR_SIZE);
130024     int status;
130025     int c;
130026     void *cache;
130027     //
130028     // Convert the table pointer to a drive number.
130029     //
130030     ptrdiff = (((intptr_t) dm_table[d].table)
130031                - ((intptr_t) ata_table));
130032     drive = ptrdiff / (sizeof (ata_t));
130033     //
130034     if (DEBUG)
```

```
130035     {
130036         k_printf ("%s: R/W=%i dev=%04x n=%ui ", __FILE__,
130037                 rw, (int) device, n);
130038         blk_cache_check ();
130039     }
130040     //
130041     // If reading, check if we already have inside
130042     // cache.
130043     //
130044     if (rw == DEV_READ)
130045     {
130046         cache = blk_cache_read (device, n);
130047         if (cache != NULL)
130048         {
130049             return (cache);
130050         }
130051     }
130052     //
130053     // Read or write.
130054     //
130055     for (c = 0, status = 0;
130056          c < count && status == 0; c++, sector++)
130057     {
130058         //
130059         if (DEBUG)
130060         {
130061             k_printf ("sec=%i ", sector);
130062         }
130063         //
130064         if (rw == DEV_READ)
130065         {
130066             status = ata_read_sector (drive, sector,
130067                                     &destination[c]);
130068         }
130069         else
130070         {
130071             status = ata_write_sector (drive, sector,
```

```
130072                                     &source[c]);
130073     }
130074 }
130075 //
130076 // If a block was read or written inside ATA
130077 // hardware, then
130078 // save it inside the cache.
130079 //
130080 if (status == 0)
130081 {
130082     cache = blk_cache_save (device, n, buffer);
130083 }
130084 else
130085 {
130086     cache = NULL;
130087 }
130088 //
130089 //
130090 //
130091 if (DEBUG)
130092 {
130093     k_printf ("\n");
130094 }
130095 //
130096 return (cache);
130097 }
```

94.2.2 kernel/blk/blk_cache_check.c



Si veda la sezione [93.3.2](#).

```
140001 #include <kernel/blk.h>
140002 #include <string.h>
140003 #include <kernel/lib_k.h>
140004 //-----
140005 void
140006 blk_cache_check (void)
```

```
140007 {
140008     int i;
140009     int j;
140010     //
140011     // check if all ages are present.
140012     //
140013     for (i = 0; i < BLK_CACHE_SIZE; i++)
140014     {
140015         if (blk_table[i].age > BLK_CACHE_MAX_AGE)
140016         {
140017             k_printf
140018                 ("blk_table[%i].age > BLK_CACHE_MAX_AGE\n", i);
140019             return;
140020         }
140021         for (j = 0; j < BLK_CACHE_SIZE; j++)
140022         {
140023             if (j != i
140024                 && blk_table[i].age == blk_table[j].age)
140025             {
140026                 k_printf
140027                     ("blk_table[%i].age == "
140028                      "blk_table[%i].age\n", i, j);
140029                 return;
140030             }
140031         }
140032     }
140033 }
```

94.2.3 kernel/blk/blk_cache_init.c

Si veda la sezione [93.3.3](#).

```
150001 #include <kernel/blk.h>
150002 //-----
150003 void
150004 blk_cache_init (void)
150005 {
```

```
150006     int i;
150007     for (i = 0; i < BLK_CACHE_SIZE; i++)
150008     {
150009         blk_table[i].age = i;      // Age is from 0 to
150010         // BLK_CACHE_MAX_AGE.
150011         blk_table[i].device = 0;
150012         blk_table[i].n = 0;
150013     }
150014 }
```

94.2.4 kernel/blk/blk_cache_read.c

«

Si veda la sezione [93.3.4](#).

```
160001 #include <kernel/blk.h>
160002 #include <string.h>
160003 //-----
160004 void *
160005 blk_cache_read (dev_t device, unsigned int n)
160006 {
160007     int i;
160008     int j;
160009     int age;
160010     //
160011     device &= 0xFFF0;
160012     //
160013     for (i = 0; i < BLK_CACHE_SIZE; i++)
160014     {
160015         if (blk_table[i].device == device
160016             && blk_table[i].n == n)
160017         {
160018             age = blk_table[i].age;
160019             for (j = 0; j < BLK_CACHE_SIZE; j++)
160020             {
160021                 if (blk_table[j].age < age)
160022                 {
160023                     blk_table[j].age++;

```

```
160024         }
160025     }
160026     blk_table[i].age = 0;
160027     //
160028     return (&blk_table[i].block);
160029 }
160030 }
160031 return (NULL);
160032 }
```

94.2.5 kernel/blk/blk_cache_save.c



Si veda la sezione [93.3.4](#).

```
170001 #include <kernel/blk.h>
170002 #include <string.h>
170003 //-----
170004 void *
170005 blk_cache_save (dev_t device, unsigned int n, void *block)
170006 {
170007     int i;
170008     int j;
170009     int age;
170010     //
170011     device &= 0xFFF0;
170012     //
170013     // Look inside the cache, if we already have
170014     // that old block.
170015     //
170016     for (i = 0; i < BLK_CACHE_SIZE; i++)
170017     {
170018         if (blk_table[i].device == device
170019             && blk_table[i].n == n)
170020         {
170021             age = blk_table[i].age;
170022             for (j = 0; j < BLK_CACHE_SIZE; j++)
170023             {
```

```
170024         if (blk_table[j].age < age)
170025             {
170026                 blk_table[j].age++;
170027             }
170028     }
170029     blk_table[i].age = 0;
170030     //
170031     // Check if the block is the same memory.
170032     //
170033     if (blk_table[i].block == block)
170034     {
170035         //
170036         // No need to transfer data.
170037         //
170038         ;
170039     }
170040     else
170041     {
170042         memcpy (blk_table[i].block, block,
170043                (size_t) BLK_SIZE);
170044     }
170045     return (&blk_table[i].block);
170046 }
170047 }
170048 //
170049 // The block is new for the cache: must find
170050 // the older and replace it.
170051 //
170052 for (i = 0; i < BLK_CACHE_SIZE; i++)
170053 {
170054     if (blk_table[i].age == BLK_CACHE_MAX_AGE)
170055     {
170056         for (j = 0; j < BLK_CACHE_SIZE; j++)
170057         {
170058             if (blk_table[j].age < BLK_CACHE_MAX_AGE)
170059                 {
170060                     blk_table[j].age++;
```



```

170061         }
170062     }
170063     blk_table[i].age = 0;
170064     blk_table[i].device = device;
170065     blk_table[i].n = n;
170066     memcpy (blk_table[i].block, block,
170067            (size_t) BLK_SIZE);
170068     //
170069     return (&blk_table[i].block);
170070 }
170071 }
170072 //
170073 // It should never happen to fail.
170074 //
170075 return (NULL);
170076 }

```

94.2.6 kernel/blk/blk_public.c

Si veda la sezione [93.3](#).

```

180001 #include <kernel/blk.h>
180002 //-----
180003 blk_cache_t blk_table[BLK_CACHE_SIZE];

```

94.3 os32: «kernel/dev.h»

Si veda la sezione [93.4](#).

```

190001 #ifndef _KERNEL_DEV_H
190002 #define _KERNEL_DEV_H 1
190003 //-----
190004 #include <sys/os32.h>
190005 #include <sys/types.h>
190006 #include <kernel/driver/ata.h>
190007 //-----

```

```

190008 #define DEV_READ          0
190009 #define DEV_WRITE        1
190010 ssize_t dev_io (pid_t pid, dev_t device, int rw,
190011                off_t offset, void *buffer,
190012                size_t size, int *eof);
190013 //-----
190014 // The following functions are used only by 'dev_io()'.
190015 //-----
190016 ssize_t dev_dm (pid_t pid, dev_t device, int rw,
190017                off_t offset, void *buffer,
190018                size_t size, int *eof);
190019 ssize_t dev_ata (pid_t pid, dev_t device, int rw,
190020                 off_t offset, void *buffer,
190021                 size_t size, int *eof);
190022 ssize_t dev_mem (pid_t pid, dev_t device, int rw,
190023                 off_t offset, void *buffer,
190024                 size_t size, int *eof);
190025 ssize_t dev_tty (pid_t pid, dev_t device, int rw,
190026                 off_t offset, void *buffer,
190027                 size_t size, int *eof);
190028 ssize_t dev_kmem (pid_t pid, dev_t device, int rw,
190029                  off_t offset, void *buffer,
190030                  size_t size, int *eof);
190031 //-----
190032 #endif

```

94.3.1	kernel/dev/dev_ata.c	1019
94.3.2	kernel/dev/dev_dm.c	1022
94.3.3	kernel/dev/dev_io.c	1023
94.3.4	kernel/dev/dev_kmem.c	1024
94.3.5	kernel/dev/dev_mem.c	1031
94.3.6	kernel/dev/dev_tty.c	1034

94.3.1 kernel/dev/dev_ata.c



Si veda la sezione [93.4.3](#).

```
200001 #include <sys/os32.h>
200002 #include <kernel/dev.h>
200003 #include <kernel/blk.h>
200004 #include <kernel/driver/ata.h>
200005 #include <kernel/lib_k.h>
200006 #include <kernel/dm.h>
200007 #include <sys/types.h>
200008 #include <ctype.h>
200009 #include <errno.h>
200010 //-----
200011 ssize_t
200012 dev_ata (pid_t pid, dev_t device, int rw, off_t offset,
200013         void *buffer, size_t size, int *eof)
200014 {
200015     ssize_t m;
200016     ssize_t n = 0;
200017     unsigned char *data_buffer = buffer;
200018     unsigned int n_blk;
200019     int i;
200020     int j = 0;
200021     char *block;
200022     //
200023     // Read the first block.
200024     //
200025     n_blk = offset / BLK_SIZE;
200026     i = offset % BLK_SIZE;
200027     //
200028     block = blk_ata (device, DEV_READ, n_blk, NULL);
200029     if (block == NULL)
200030     {
200031         errset (errno);
200032         return ((ssize_t) - 1);
200033     }
200034     //
```

```
200035 // Read or write.
200036 //
200037 if (rw == DEV_READ)
200038 {
200039     while (size)
200040     {
200041         for (;
200042             i < BLK_SIZE && size > 0;
200043             i++, j++, n++, size--, offset++)
200044         {
200045             data_buffer[j] = block[i];
200046         }
200047         if (size)
200048         {
200049             n_blk = offset / BLK_SIZE;
200050             i = offset % BLK_SIZE;
200051             block = blk_ata (device, DEV_READ, n_blk,
200052                             NULL);
200053             if (block == NULL)
200054             {
200055                 errset (errno);
200056                 return (n); // Up to the last
200057                             // block read.
200058             }
200059         }
200060     }
200061 }
200062 else
200063 {
200064     //
200065     // Write.
200066     //
200067     while (size)
200068     {
200069         //
200070         // The last block was written, so update
200071         // the counter 'm'.
```

```
200072         //
200073         m = n;
200074         //
200075         for (;
200076             i < BLK_SIZE && size > 0;
200077             i++, j++, n++, size--, offset++)
200078             {
200079                 block[i] = data_buffer[j];
200080             }
200081         block = blk_ata (device, DEV_WRITE, n_blk, block);
200082         if (block == NULL)
200083             {
200084                 errset (errno);
200085                 return (m);          // Up to the last
200086                 // block written.
200087             }
200088         if (size)
200089             {
200090                 n_blk = offset / BLK_SIZE;
200091                 i = offset % BLK_SIZE;
200092                 block = blk_ata (device, DEV_READ, n_blk,
200093                                 NULL);
200094                 if (block == NULL)
200095                     {
200096                         errset (errno);
200097                         return (m);    // Up to the last
200098                         // block written.
200099                     }
200100             }
200101     }
200102 }
200103 //
200104 // Everything was right, so 'n' is valid for both
200105 // read or write.
200106 //
200107 return (n);
```

200108	}
--------	---

94.3.2 kernel/dev/dev_dm.c

<<

Si veda la sezione [93.4.2](#).

```
210001 #include <sys/os32.h>
210002 #include <kernel/dev.h>
210003 #include <kernel/driver/ata.h>
210004 #include <kernel/lib_k.h>
210005 #include <kernel/dm.h>
210006 #include <sys/types.h>
210007 #include <ctype.h>
210008 #include <errno.h>
210009 //-----
210010 ssize_t
210011 dev_dm (pid_t pid, dev_t device, int rw, off_t offset,
210012         void *buffer, size_t size, int *eof)
210013 {
210014     int dev_minor = minor (device);
210015     int p = dev_minor & 0x000F;
210016     int d = ((dev_minor & 0x00F0) >> 4);
210017     //
210018     // If it is a partition, must verify if such
210019     // partition exists.
210020     //
210021     if (p)
210022     {
210023         //
210024         // It is a partition.
210025         //
210026         if (dm_table[d].part[p].type == PART_TYPE_NONE)
210027         {
210028             errset (ENODEV);
210029             return ((ssize_t) - 1);
210030         }
210031     }
```

```

210032 //
210033 // Shift the offset to the start of partition.
210034 //
210035 offset += (dm_table[d].part[p].start);
210036 //
210037 // Call the right hardware driver.
210038 //
210039 switch (dm_table[d].type)
210040 {
210041     case DM_TYPE_ATA:
210042         return (dev_ata
210043                 (pid, device, rw, offset, buffer, size, eof));
210044         break;
210045     default:
210046         errset (ENODEV);
210047         return ((ssize_t) - 1);
210048 }
210049 }

```

94.3.3 kernel/dev/dev_io.c

Si veda la sezione [93.4.1](#).

```

220001 #include <sys/os32.h>
220002 #include <kernel/dev.h>
220003 #include <sys/types.h>
220004 #include <errno.h>
220005 #include <kernel/ibm_i386.h>
220006 #include <kernel/proc.h>
220007 #include <string.h>
220008 #include <signal.h>
220009 #include <kernel/lib_k.h>
220010 #include <ctype.h>
220011 #include <kernel/driver/tty.h>
220012 //-----
220013 ssize_t
220014 dev_io (pid_t pid, dev_t device, int rw, off_t offset,

```



```
220015     void *buffer, size_t size, int *eof)
220016 {
220017     int dev_major = major (device);
220018     if (rw != DEV_READ && rw != DEV_WRITE)
220019     {
220020         errset (EIO);
220021         return (-1);
220022     }
220023     switch (dev_major)
220024     {
220025     case DEV_MEM_MAJOR:
220026         return (dev_mem
220027                 (pid, device, rw, offset, buffer, size, eof));
220028     case DEV_TTY_MAJOR:
220029         return (dev_tty
220030                 (pid, device, rw, offset, buffer, size, eof));
220031     case DEV_CONSOLE_MAJOR:
220032         return (dev_tty
220033                 (pid, device, rw, offset, buffer, size, eof));
220034     case DEV_DM_MAJOR:
220035         return (dev_dm
220036                 (pid, device, rw, offset, buffer, size, eof));
220037     case DEV_KMEM_MAJOR:
220038         return (dev_kmem
220039                 (pid, device, rw, offset, buffer, size, eof));
220040     default:
220041         errset (ENODEV);
220042         return (-1);
220043     }
220044 }
```

94.3.4 kernel/dev/dev_kmem.c



Si veda la sezione [93.4.4](#).

```
230001 #include <sys/os32.h>
230002 #include <kernel/dev.h>
```



```
230003 #include <sys/types.h>
230004 #include <errno.h>
230005 #include <kernel/memory.h>
230006 #include <kernel/proc.h>
230007 #include <kernel/net/arp.h>
230008 #include <kernel/net/route.h>
230009 #include <kernel/net.h>
230010 #include <string.h>
230011 #include <signal.h>
230012 #include <ctype.h>
230013 //-----
230014 ssize_t
230015 dev_kmem (pid_t pid, dev_t device, int rw,
230016          off_t offset, void *buffer, size_t size, int *eof)
230017 {
230018     inode_t *inode;
230019     sb_t *sb;
230020     file_t *file;
230021     void *start;
230022     char *m;
230023     //
230024     // Only read is allowed.
230025     //
230026     if (rw != DEV_READ)
230027     {
230028         errset (EIO);      // I/O error.
230029         return ((ssize_t) - 1);
230030     }
230031     //
230032     // Only positive offset is allowed.
230033     //
230034     if (offset < 0)
230035     {
230036         errset (EIO);      // I/O error.
230037         return ((ssize_t) - 1);
230038     }
230039     //
```

```
230040 // Read is selected (and is the only access
230041 // allowed).
230042 //
230043 switch (device)
230044 {
230045     case DEV_KMEM_PS:
230046         //
230047         // Verify if the selected slot can be read.
230048         //
230049         if (offset >= PROCESS_MAX)
230050             {
230051                 errset (EIO); // I/O error.
230052                 return ((ssize_t) - 1);
230053             }
230054         //
230055         // Correct the size to be read.
230056         //
230057         if (sizeof (proc_t) < size)
230058             {
230059                 size = sizeof (proc_t);
230060             }
230061         //
230062         // Get the pointer to the selected slot.
230063         //
230064         start = proc_reference ((pid_t) offset);
230065         break;
230066     case DEV_KMEM_MMP:
230067         //
230068         // Correct the size to be read.
230069         //
230070         if (offset >= (MEM_MAX_BLOCKS / 8))
230071             {
230072                 *eof = 1;
230073                 errset (EIO); // I/O error.
230074                 return ((ssize_t) - 1);
230075             }
230076         //
```

```
230077 // Reduce size if necessary.
230078 //
230079 if ((offset + size) > (MEM_MAX_BLOCKS / 8))
230080 {
230081     size = ((MEM_MAX_BLOCKS / 8) - offset);
230082 }
230083 //
230084 // Get the pointer to the map: offset is not
230085 // taken
230086 // into consideration.
230087 //
230088 m = (char *) mb_reference ();
230089 //
230090 start = &m[offset];
230091 //
230092 break;
230093 case DEV_KMEM_SB:
230094 //
230095 // Verify if the selected slot can be read.
230096 //
230097 if (offset >= SB_MAX_SLOTS)
230098 {
230099     errset (EIO); // I/O error.
230100     return ((ssize_t) - 1);
230101 }
230102 //
230103 // Get a reference to the super block table.
230104 //
230105 sb = sb_reference (0);
230106 //
230107 // Correct the size to be read.
230108 //
230109 if (sizeof (sb_t) < size)
230110 {
230111     size = sizeof (sb_t);
230112 }
230113 //
```

```
230114 // Get the pointer to the selected super block
230115 // slot.
230116 //
230117 start = &sb[offset];
230118 break;
230119 case DEV_KMEM_INODE:
230120 //
230121 // Verify if the selected slot can be read.
230122 //
230123 if (offset >= INODE_MAX_SLOTS)
230124 {
230125     errset (EIO); // I/O error.
230126     return ((ssize_t) - 1);
230127 }
230128 //
230129 // Get a reference to the inode table.
230130 //
230131 inode = inode_reference (0, 0);
230132 //
230133 // Correct the size to be read.
230134 //
230135 if (sizeof (inode_t) < size)
230136 {
230137     size = sizeof (inode_t);
230138 }
230139 //
230140 // Get the pointer to the selected inode slot.
230141 //
230142 start = &inode[offset];
230143 break;
230144 case DEV_KMEM_FILE:
230145 //
230146 // Verify if the selected slot can be read.
230147 //
230148 if (offset >= FILE_MAX_SLOTS)
230149 {
230150     errset (EIO); // I/O error.
```

```
230151         return ((ssize_t) - 1);
230152     }
230153     //
230154     // Get a reference to the file table.
230155     //
230156     file = file_reference (0);
230157     //
230158     // Correct the size to be read.
230159     //
230160     if (sizeof (file_t) < size)
230161     {
230162         size = sizeof (file_t);
230163     }
230164     //
230165     // Get the pointer to the selected inode slot.
230166     //
230167     start = &file[offset];
230168     break;
230169 case DEV_KMEM_ARP:
230170     //
230171     // Verify if the selected slot can be read.
230172     //
230173     if (offset >= ARP_MAX_ITEMS)
230174     {
230175         errset (EIO); // I/O error.
230176         return ((ssize_t) - 1);
230177     }
230178     //
230179     // Correct the size to be read.
230180     //
230181     if (sizeof (arp_t) < size)
230182     {
230183         size = sizeof (arp_t);
230184     }
230185     //
230186     // Get the pointer to the selected ARP item.
230187     //
```

```
230188     start = &arp_table[offset];
230189     break;
230190 case DEV_KMEM_NET:
230191     //
230192     // Verify if the selected slot can be read.
230193     //
230194     if (offset >= NET_MAX_DEVICES)
230195     {
230196         errset (EIO); // I/O error.
230197         return ((ssize_t) - 1);
230198     }
230199     //
230200     // Correct the size to be read.
230201     //
230202     if (sizeof (net_t) < size)
230203     {
230204         size = sizeof (net_t);
230205     }
230206     //
230207     // Get the pointer to the selected NET table
230208     // item.
230209     //
230210     start = &net_table[offset];
230211     break;
230212 case DEV_KMEM_ROUTE:
230213     //
230214     // Verify if the selected slot can be read.
230215     //
230216     if (offset >= ROUTE_MAX_ROUTES)
230217     {
230218         errset (EIO); // I/O error.
230219         return ((ssize_t) - 1);
230220     }
230221     //
230222     // Correct the size to be read.
230223     //
230224     if (sizeof (route_t) < size)
```

```
230225     {
230226         size = sizeof (route_t);
230227     }
230228     //
230229     // Get the pointer to the selected NET table
230230     // item.
230231     //
230232     start = &route_table[offset];
230233     break;
230234     default:
230235         errset (ENODEV); // No such device.
230236         return ((ssize_t) - 1);
230237     }
230238     //
230239     // At this point, data is ready to be copied to the
230240     // buffer.
230241     //
230242     memcpy (buffer, start, size);
230243     //
230244     // Return size read.
230245     //
230246     return (size);
230247 }
```

94.3.5 kernel/dev/dev_mem.c

Si veda la sezione [93.4.5](#).

```
240001 #include <sys/os32.h>
240002 #include <kernel/dev.h>
240003 #include <sys/types.h>
240004 #include <errno.h>
240005 #include <kernel/memory.h>
240006 #include <kernel/ibm_i386.h>
240007 #include <kernel/proc.h>
240008 #include <string.h>
240009 #include <signal.h>
```

```
240010 #include <kernel/lib_k.h>
240011 #include <ctype.h>
240012 //-----
240013 ssize_t
240014 dev_mem (pid_t pid, dev_t device, int rw, off_t offset,
240015         void *buffer, size_t size, int *eof)
240016 {
240017     uint8_t *buffer08 = (uint8_t *) buffer;
240018     uint16_t *buffer16 = (uint16_t *) buffer;
240019     ssize_t n;
240020
240021     if (device == DEV_MEM)           // DEV_MEM
240022     {
240023         if (rw == DEV_READ)
240024         {
240025             memcpy (buffer, (void *) (int) offset, size);
240026             n = size;
240027         }
240028         else
240029         {
240030             if (pid == 0)
240031             {
240032                 memcpy ((void *) (int) offset, buffer, size);
240033                 n = size;
240034             }
240035             else
240036             {
240037                 k_printf
240038                 ("kernel alert: only the kernel "
240039                  "can write the memory where it "
240040                  "likes!\n");
240041                 errset (EIO);           // I/O error.
240042                 return ((ssize_t) - 1);
240043             }
240044         }
240045     }
240046     else if (device == DEV_NULL)     // DEV_NULL
```



```
240047     {
240048         n = 0;
240049     }
240050     else if (device == DEV_ZERO)    // DEV_ZERO
240051     {
240052         if (rw == DEV_READ)
240053         {
240054             for (n = 0; n < size; n++)
240055             {
240056                 buffer08[n] = 0;
240057             }
240058         }
240059         else
240060         {
240061             n = 0;
240062         }
240063     }
240064     else if (device == DEV_PORT)    // DEV_PORT
240065     {
240066         if (rw == DEV_READ)
240067         {
240068             if (size == 1)
240069             {
240070                 buffer08[0] = in_8 (offset);
240071                 n = 1;
240072             }
240073             else if (size == 2)
240074             {
240075                 buffer16[0] = in_16 (offset);
240076                 n = 2;
240077             }
240078             else
240079             {
240080                 n = 0;
240081             }
240082         }
240083     else
```

```
240084     {
240085         if (size == 1)
240086         {
240087             out_8 (offset, buffer08[0]);
240088         }
240089         else if (size == 2)
240090         {
240091             out_16 (offset, buffer16[0]);
240092             n = 2;
240093         }
240094         else
240095         {
240096             n = 0;
240097         }
240098     }
240099 }
240100 else
240101 {
240102     errset (ENODEV);
240103     return ((ssize_t) - 1);
240104 }
240105 return (n);
240106 }
```

94.3.6 kernel/dev/dev_tty.c

«

Si veda la sezione [93.4.6](#).

```
250001 #include <sys/os32.h>
250002 #include <kernel/dev.h>
250003 #include <sys/types.h>
250004 #include <errno.h>
250005 #include <kernel/memory.h>
250006 #include <kernel/ibm_i386.h>
250007 #include <kernel/proc.h>
250008 #include <string.h>
250009 #include <signal.h>
```

```
250010 #include <kernel/lib_k.h>
250011 #include <ctype.h>
250012 #include <kernel/driver/tty.h>
250013 //-----
250014 ssize_t
250015 dev_tty (pid_t pid, dev_t device, int rw, off_t offset,
250016         void *buffer, size_t size, int *eof)
250017 {
250018     uint8_t *buffer08 = (uint8_t *) buffer;
250019     ssize_t n;
250020     proc_t *ps;
250021     int key;
250022     //
250023     // Get process. Variable 'ps' will be 'NULL' if the
250024     // process ID is
250025     // not valid.
250026     //
250027     ps = proc_reference (pid);
250028     //
250029     // Convert 'DEV_TTY' with the controlling terminal
250030     // for the process.
250031     //
250032     if (device == DEV_TTY)
250033     {
250034         device = ps->device_tty;
250035         //
250036         // As a last resort, use the generic
250037         // 'DEV_CONSOLE'.
250038         //
250039         if (device == DEV_UNDEFINED || device == DEV_TTY)
250040         {
250041             device = DEV_CONSOLE;
250042         }
250043     }
250044     //
250045     // Convert 'DEV_CONSOLE' to the currently active
250046     // console.
```

```
250047 //
250048 if (device == DEV_CONSOLE)
250049 {
250050     device = tty_console ((dev_t) 0);
250051     //
250052     // As a last resort, use the first console:
250053     // 'DEV_CONSOLE0'.
250054     //
250055     if (device == DEV_UNDEFINED || device == DEV_TTY)
250056     {
250057         device = DEV_CONSOLE0;
250058     }
250059 }
250060 //
250061 // Read or write.
250062 //
250063 if (rw == DEV_READ)
250064 {
250065     for (n = 0; n < size; n++)
250066     {
250067         key = tty_read (device);
250068         if (key == 0 && n == 0)
250069         {
250070             //
250071             // A single line contains zero: this is
250072             // made by a VEOF
250073             // character (^d), that is, the input is
250074             // closed,
250075             // so return zero read and EOF.
250076             //
250077             *eof = 1;
250078             return (0);
250079         }
250080         else if (key == -1 && n == 0)
250081         {
250082             //
250083             // At the moment, there is just nothing
```

```
250084         // to read.
250085         //
250086         errset (EAGAIN);
250087         return (-1);
250088     }
250089     else if (key == -1 && n > 0)
250090     {
250091         //
250092         // Finished to read.
250093         //
250094         break;
250095     }
250096     else
250097     {
250098         buffer08[n] = key;
250099     }
250100 }
250101 }
250102 else
250103 {
250104     for (n = 0; n < size; n++)
250105     {
250106         tty_write (device, (int) buffer08[n]);
250107     }
250108 }
250109 return (n);
250110 }
```

94.4 os32: «kernel/dm.h»

Si veda la sezione [93.5](#).

```
260001 #ifndef _KERNEL_DM_H
260002 #define _KERNEL_DM_H    1
260003 //-----
260004 #include <stdint.h>
260005 #include <sys/types.h>
```

```

260006 #include <kernel/part.h>
260007 //-----
260008 #define DM_MAX_DEVICES      4
260009 #define DM_TYPE_NONE       0
260010 #define DM_TYPE_ATA        1
260011 //
260012 typedef struct
260013 {
260014     int type;
260015     void *table;
260016     struct
260017     {
260018         off_t start;
260019         size_t size;
260020         uint8_t type;
260021     } part[PART_MAX + 1];
260022 } dm_t;
260023 //
260024 extern dm_t dm_table[DM_MAX_DEVICES];
260025 //-----
260026 void dm_init (void);
260027 //-----
260028 #endif

```

94.4.1	kernel/dm/dm_init.c	1041
94.4.2	kernel/dm/dm_public.c	1044
94.4.3	kernel/driver/ata.h	1044
94.4.4	kernel/driver/ata/ata_cmd_identify_device.c	1049
94.4.5	kernel/driver/ata/ata_cmd_read_sectors.c	1051
94.4.6	kernel/driver/ata/ata_cmd_write_sectors.c	1053
94.4.7	kernel/driver/ata/ata_device.c	1055
94.4.8	kernel/driver/ata/ata_drq.c	1057

94.4.9	kernel/driver/ata/ata_init.c	1059
94.4.10	kernel/driver/ata/ata_lba28.c	1066
94.4.11	kernel/driver/ata/ata_public.c	1067
94.4.12	kernel/driver/ata/ata_rdy.c	1068
94.4.13	kernel/driver/ata/ata_reset.c	1070
94.4.14	kernel/driver/ata/ata_valid.c	1070
94.4.15	kernel/driver/kbd.h	1071
94.4.16	kernel/driver/kbd/kbd_isr.c	1072
94.4.17	kernel/driver/kbd/kbd_load.c	1077
94.4.18	kernel/driver/kbd/kbd_public.c	1081
94.4.19	kernel/driver/nic/ne2k.h	1081
94.4.20	kernel/driver/nic/ne2k/ne2k_check.c	1085
94.4.21	kernel/driver/nic/ne2k/ne2k_isr.c	1088
94.4.22	kernel/driver/nic/ne2k/ne2k_isr_expect.c	1090
94.4.23	kernel/driver/nic/ne2k/ne2k_reset.c	1092
94.4.24	kernel/driver/nic/ne2k/ne2k_rx.c	1104
94.4.25	kernel/driver/nic/ne2k/ne2k_rx_reset.c	1113
94.4.26	kernel/driver/nic/ne2k/ne2k_tx.c	1115
94.4.27	kernel/driver/pci.h	1119
94.4.28	kernel/driver/pci/pci_init.c	1122
94.4.29	kernel/driver/pci/pci_public.c	1125
94.4.30	kernel/driver/screen.h	1125

94.4.31	kernel/driver/screen/screen_clear.c	1127
94.4.32	kernel/driver/screen/screen_current.c	1128
94.4.33	kernel/driver/screen/screen_init.c	1128
94.4.34	kernel/driver/screen/screen_new_line.c	1129
94.4.35	kernel/driver/screen/screen_number.c	1130
94.4.36	kernel/driver/screen/screen_pointer.c	1132
94.4.37	kernel/driver/screen/screen_public.c	1132
94.4.38	kernel/driver/screen/screen_putc.c	1133
94.4.39	kernel/driver/screen/screen_scroll.c	1134
94.4.40	kernel/driver/screen/screen_select.c	1136
94.4.41	kernel/driver/screen/screen_update.c	1137
94.4.42	kernel/driver/tty.h	1139
94.4.43	kernel/driver/tty/tty_console.c	1140
94.4.44	kernel/driver/tty/tty_init.c	1141
94.4.45	kernel/driver/tty/tty_public.c	1143
94.4.46	kernel/driver/tty/tty_read.c	1144
94.4.47	kernel/driver/tty/tty_reference.c	1145
94.4.48	kernel/driver/tty/tty_write.c	1146

94.4.1 kernel/dm/dm_init.c



Si veda la sezione [93.5](#).

```
270001 #include <kernel/dm.h>
270002 #include <kernel/part.h>
270003 #include <kernel/driver/ata.h>
270004 #include <kernel/lib_k.h>
270005 #include <stdint.h>
270006 #include <errno.h>
270007 //-----
270008 void
270009 dm_init (void)
270010 {
270011     int d;
270012     int a;
270013     int p;
270014     ata_sector_t sector_buffer;
270015     part_t *part;
270016     int status;
270017     //
270018     // Reset the data-memory table.
270019     //
270020     for (d = 0; d < DM_MAX_DEVICES; d++)
270021     {
270022         dm_table[d].type = DM_TYPE_NONE;
270023         dm_table[d].table = NULL;
270024         dm_table[d].part[0].start = 0;
270025         dm_table[d].part[0].size = 0;
270026         dm_table[d].part[0].type = PART_TYPE_NO_PART;
270027         for (p = 0; p < PART_MAX; p++)
270028         {
270029             dm_table[d].part[p + 1].start = 0;
270030             dm_table[d].part[p + 1].size = 0;
270031             dm_table[d].part[p + 1].type = PART_TYPE_NONE;
270032         }
270033     }
270034     //
```

```
270035 // Reset data-memory index.
270036 //
270037 d = 0;
270038 //
270039 // Init ATA devices.
270040 //
270041 ata_init ();
270042 //
270043 // Assign ATA devices to the first data-memory
270044 // items.
270045 //
270046 for (a = 0; a < ATA_MAX_DEVICES; a++)
270047 {
270048     if (ata_table[a].present == 0)
270049     {
270050         //
270051         // Current data-memory device will be
270052         // used for the next ATA device, if any.
270053         //
270054         continue;
270055     }
270056     //
270057     // Show something.
270058     //
270059     k_printf ("%s] ATA drive=%i total sectors=%i\n",
270060               __func__, a, (int) ata_table[a].sectors);
270061     //
270062     dm_table[d].type = DM_TYPE_ATA;
270063     dm_table[d].table = &ata_table[a];
270064     dm_table[d].part[0].start = 0;
270065     dm_table[d].part[0].size = ata_table[a].sectors;
270066     dm_table[d].part[0].type = PART_TYPE_NO_PART;
270067     //
270068     // Read partitions.
270069     //
270070     status = ata_read_sector (a, 0, &sector_buffer);
270071     //
```

```
270072     if (status)
270073     {
270074         errset (errno);
270075         k_perror (NULL);
270076     }
270077     else
270078     {
270079         part =
270080             ((void *) &sector_buffer) + PART_TABLE_OFF;
270081         //
270082         for (p = 0; p < PART_MAX; p++)
270083         {
270084             //
270085             dm_table[d].part[p + 1].start =
270086                 part->l_start * ATA_SECTOR_SIZE;
270087             dm_table[d].part[p + 1].size =
270088                 part->size * ATA_SECTOR_SIZE;
270089             dm_table[d].part[p + 1].type = part->type;
270090             //
270091             // Show info.
270092             //
270093             if (part->type != 0)
270094             {
270095                 k_printf ("%s] partition type=%02x "
270096                     "start sector=%i "
270097                     "total sectors=%i\n",
270098                     __func__, (int) part->type,
270099                     (int) part->l_start,
270100                     (int) part->size);
270101             }
270102             //
270103             part++;
270104         }
270105     }
270106     //
270107     // Next data-memory device.
270108     //
```

```

270109     d++;
270110     }
270111 }

```

94.4.2 kernel/dm/dm_public.c

<<

Si veda la sezione [93.5](#).

```

280001 #include <kernel/dm.h>
280002 //-----
280003 dm_t dm_table[DM_MAX_DEVICES];

```

94.4.3 kernel/driver/ata.h

<<

Si veda la sezione [93.2](#).

```

290001 #ifndef _KERNEL_DRIVER_ATA_H
290002 #define _KERNEL_DRIVER_ATA_H    1
290003 //-----
290004 #include <stdint.h>
290005 #include <sys/types.h>
290006 #include <time.h>
290007 //-----
290008 //
290009 // I/O ports, used to access ATA bus registers. These
290010 // I/O ports are different for every ATA bus.
290011 //
290012 #define ATA0_DATA           0x1F0        // r/w
290013 #define ATA0_FEATURE       0x1F1        // -/w
290014 #define ATA0_ERROR        0x1F1        // r/-
290015 #define ATA0_COUNT        0x1F2        // r/w
290016 #define ATA0_LOW          0x1F3        // r/w
290017 #define ATA0_MID          0x1F4        // r/w
290018 #define ATA0_HIGH         0x1F5        // r/w
290019 #define ATA0_DEVICE       0x1F6        // r/w
290020 #define ATA0_COMMAND      0x1F7        // -/w
290021 #define ATA0_STATUS       0x1F7        // r/- regular

```

```
290022 // status
290023 #define ATA0_CONTROL 0x3F6 // -/w
290024 #define ATA0_ALTERNATE 0x3F6 // w/-
290025 // alternate
290026 // status
290027 //
290028 #define ATA1_DATA 0x170 // r/w
290029 #define ATA1_FEATURE 0x171 // -/w
290030 #define ATA1_ERROR 0x171 // r/-
290031 #define ATA1_COUNT 0x172 // r/w
290032 #define ATA1_LOW 0x173 // r/w
290033 #define ATA1_MID 0x174 // r/w
290034 #define ATA1_HIGH 0x175 // r/w
290035 #define ATA1_DEVICE 0x176 // r/w
290036 #define ATA1_COMMAND 0x177 // -/w
290037 #define ATA1_STATUS 0x177 // r/- regular
290038 // status
290039 #define ATA1_CONTROL 0x376 // -/w
290040 #define ATA1_ALTERNATE 0x376 // w/-
290041 // alternate
290042 // status
290043 //
290044 #define ATA2_DATA 0x1E8 // r/w
290045 #define ATA2_FEATURE 0x1E9 // -/w
290046 #define ATA2_ERROR 0x1E9 // r/-
290047 #define ATA2_COUNT 0x1EA // r/w
290048 #define ATA2_LOW 0x1EB // r/w
290049 #define ATA2_MID 0x1EC // r/w
290050 #define ATA2_HIGH 0x1ED // r/w
290051 #define ATA2_DEVICE 0x1EE // r/w
290052 #define ATA2_COMMAND 0x1EF // -/w
290053 #define ATA2_STATUS 0x1EF // r/- regular
290054 // status
290055 #define ATA2_CONTROL 0x3E6 // -/w
290056 #define ATA2_ALTERNATE 0x3E6 // w/-
290057 // alternate
290058 // status
```

```

290059 //
290060 #define ATA3_DATA          0x168      // r/w
290061 #define ATA3_FEATURE       0x169      // -/w
290062 #define ATA3_ERROR         0x169      // r/-
290063 #define ATA3_COUNT         0x16A      // r/w
290064 #define ATA3_LOW           0x16B      // r/w
290065 #define ATA3_MID           0x16C      // r/w
290066 #define ATA3_HIGH          0x16D      // r/w
290067 #define ATA3_DEVICE        0x16E      // r/w
290068 #define ATA3_COMMAND       0x16F      // -/w
290069 #define ATA3_STATUS        0x16F      // r/- regular
290070                                     // status
290071 #define ATA3_CONTROL        0x366      // -/w
290072 #define ATA3_ALTERNATE     0x366      // w/-
290073                                     // alternate
290074                                     // status
290075 //
290076 // Status register flags (regular or alternate).
290077 //
290078 #define ATA_STATUS_BSY      0x80      // Busy
290079 #define ATA_STATUS_DRDY     0x40      // Ready
290080 #define ATA_STATUS_DF       0x20      // Drive Fault
290081 #define ATA_STATUS_DRQ      0x08      // Data
290082                                     // request
290083 #define ATA_STATUS_ERR      0x01      // Error
290084 //
290085 // Values to put to the device register
290086 //
290087 #define ATA_DEVICE_CHS       0x00
290088 #define ATA_DEVICE_LBA       0x40
290089 #define ATA_DEVICE_MASTER    0x00
290090 #define ATA_DEVICE_SLAVE     0x10
290091 //
290092 // Values to put to the command register
290093 //
290094 #define ATA_COMMAND_IDENTIFY_DEVICE  0xEC
290095 #define ATA_COMMAND_READ_SECTORS     0x20

```

```
290096 #define ATA_COMMAND_WRITE_SECTORS      0x30
290097 #define ATA_COMMAND_FLUSH_CACHE      0xE7
290098 //
290099 // Values to put to the control register
290100 // (device control register).
290101 //
290102 #define ATA_CONTROL_HOB                0x80
290103 #define ATA_CONTROL_SRST                0x04    // Software
290104 // reset.
290105 #define ATA_CONTROL_NIEN                0x01    // No
290106 // Interrupt
290107 // enabled.
290108 //
290109 //
290110 //
290111 #define ATA_MAX_DEVICES                 8        // Fixed.
290112 #define ATA_SECTOR_SIZE                 512     // Fixed.
290113 #define ATA_TIMEOUT \
290114     ((clock_t) (CLOCKS_PER_SEC * 1))    // 1 s
290115 #define ATA_TIMEOUT_FLUSH \
290116     ((clock_t) (CLOCKS_PER_SEC * 10))    // 10 s
290117 //
290118 //
290119 //
290120 typedef struct
290121 {
290122     unsigned short r_data;
290123     unsigned short r_feature;
290124     unsigned short r_error;
290125     unsigned short r_count;
290126     unsigned short r_low;
290127     unsigned short r_mid;
290128     unsigned short r_high;
290129     unsigned short r_device;
290130     unsigned short r_command;
290131     unsigned short r_status;
290132     unsigned short r_control;
```

```
290133     unsigned short r_alternate;
290134     unsigned char bus;
290135     unsigned char target;
290136     unsigned char present;
290137     uint16_t id[ATA_SECTOR_SIZE / 2];
290138     unsigned int sectors;
290139 } ata_t;
290140 //
290141 typedef struct
290142 {
290143     char byte[ATA_SECTOR_SIZE];
290144 } ata_sector_t;
290145 //
290146 extern ata_t ata_table[ATA_MAX_DEVICES];
290147 //-----
290148 void ata_init (void);
290149 void ata_reset (int drive);
290150 int ata_valid (int drive);
290151 //-----
290152 int ata_cmd_identify_device (int drive, void *buffer);
290153 int ata_cmd_read_sectors (int drive,
290154                          unsigned int sector,
290155                          unsigned char count,
290156                          void *buffer);
290157 int ata_cmd_write_sectors (int drive,
290158                          unsigned int sector,
290159                          unsigned char count,
290160                          void *buffer);
290161 int ata_device (int drive, unsigned int sector);
290162 //-----
290163 int ata_rdy (int drive, clock_t timeout);
290164 int ata_drq (int drive, clock_t timeout);
290165 int ata_lba28 (int drive, unsigned int sector,
290166              unsigned char count);
290167 //-----
290168 #define ata_read_sector(drive, sector, buffer) \
290169     (ata_cmd_read_sectors ((int) drive, \
```



```
290170     (unsigned int) sector, \  
290171     (unsigned char) 1, (void *) buffer))  
290172 //  
290173 #define ata_write_sector(drive, sector, buffer) \  
290174     (ata_cmd_write_sectors ((int) drive, \  
290175     (unsigned int) sector, \  
290176     (unsigned char) 1, (void *) buffer))  
290177 //-----  
290178 #endif
```

94.4.4 kernel/driver/ata/ata_cmd_identify_device.c

<<

Si veda la sezione [93.2](#).

```
300001 #include <kernel/driver/ata.h>  
300002 #include <kernel/lib_k.h>  
300003 #include <kernel/ibm_i386.h>  
300004 #include <stdint.h>  
300005 #include <errno.h>  
300006 //-----  
300007 int  
300008 ata_cmd_identify_device (int drive, void *buffer)  
300009 {  
300010     unsigned char status;  
300011     int s;  
300012     int i;  
300013     uint16_t *id = buffer;  
300014     //  
300015     // Register 'device'.  
300016     //  
300017     s = ata_device (drive, 0);  
300018     if (s < 0)  
300019     {  
300020         errset (errno);  
300021         return (-1);  
300022     }  
300023     //
```

```
300024 // Send 'command'
300025 //
300026 out_8 (ata_table[drive].r_command,
300027       ATA_COMMAND_IDENTIFY_DEVICE);
300028 //
300029 // Read the regular status port.
300030 //
300031 status = in_8 (ata_table[drive].r_status);
300032 //
300033 // If the status is zero, there is no drive.
300034 //
300035 if (status == 0)
300036 {
300037     //
300038     // Clear the 'id[]' array and return.
300039     //
300040     for (i = 0; i < (ATA_SECTOR_SIZE / 2); i++)
300041     {
300042         id[i] = 0;
300043     }
300044     return (0);
300045 }
300046 //
300047 // Wait for the drive ready to send data.
300048 //
300049 s = ata_drq (drive, ATA_TIMEOUT);
300050 if (s < 0)
300051 {
300052     errset (errno);
300053     return (-1);
300054 }
300055 //
300056 // Read data.
300057 //
300058 for (i = 0; i < (ATA_SECTOR_SIZE / 2); i++)
300059 {
300060     id[i] = in_16 (ata_table[drive].r_data);
```

```
300061     }
300062     //
300063     // Return.
300064     //
300065     return (0);
300066 }
```

94.4.5 kernel/driver/ata/ata_cmd_read_sectors.c



Si veda la sezione [93.2](#).

```
310001 #include <kernel/driver/ata.h>
310002 #include <kernel/lib_k.h>
310003 #include <kernel/ibm_i386.h>
310004 #include <stdint.h>
310005 #include <errno.h>
310006 //-----
310007 int
310008 ata_cmd_read_sectors (int drive, unsigned int sector,
310009                      unsigned char count, void *buffer)
310010 {
310011     int s;
310012     int i;
310013     int c;
310014     uint16_t *destination = buffer;
310015     //
310016     // Set LBA 28 parameters.
310017     //
310018     s = ata_lba28 (drive, sector, count);
310019     if (s < 0)
310020     {
310021         errset (errno);
310022         return (-1);
310023     }
310024     //
310025     // Send 'command'
310026     //
```

```
310027 out_8 (ata_table[drive].r_command,
310028         ATA_COMMAND_READ_SECTORS);
310029 //
310030 // Parameter 'count' equal to zero means 256
310031 // sectors.
310032 //
310033 if (count == 0)
310034     {
310035         c = 256;
310036     }
310037 else
310038     {
310039         c = count;
310040     }
310041 //
310042 // Read 'c' sectors.
310043 //
310044 for (; c > 0; c--)
310045     {
310046         s = ata_drq (drive, ATA_TIMEOUT);
310047         if (s < 0)
310048             {
310049                 errset (errno);
310050                 return (-1);
310051             }
310052 //
310053 // Read sector.
310054 //
310055         for (i = 0; i < (ATA_SECTOR_SIZE / 2); i++)
310056             {
310057                 destination[i] = in_16 (ata_table[drive].r_data);
310058             }
310059     }
310060 //
310061 // Wait that the device returns ready.
310062 //
310063 s = ata_rdy (drive, ATA_TIMEOUT);
```

```
310064     if (s < 0)
310065     {
310066         errset (errno);
310067         return (-1);
310068     }
310069     //
310070     // Return.
310071     //
310072     return (0);
310073 }
```

94.4.6 kernel/driver/ata/ata_cmd_write_sectors.c



Si veda la sezione [93.2](#).

```
320001 #include <kernel/driver/ata.h>
320002 #include <kernel/lib_k.h>
320003 #include <kernel/ibm_i386.h>
320004 #include <stdint.h>
320005 #include <errno.h>
320006 //-----
320007 int
320008 ata_cmd_write_sectors (int drive, unsigned int sector,
320009                       unsigned char count, void *buffer)
320010 {
320011     int s;
320012     int i;
320013     int c;
320014     uint16_t *source = buffer;
320015     //
320016     // Set LBA 28 parameters.
320017     //
320018     s = ata_lba28 (drive, sector, count);
320019     if (s < 0)
320020     {
320021         errset (errno);
320022         return (-1);
```

```
320023     }
320024     //
320025     // Send 'command'
320026     //
320027     out_8 (ata_table[drive].r_command,
320028           ATA_COMMAND_WRITE_SECTORS);
320029     //
320030     // Parameter 'count' equal to zero means 256
320031     // sectors.
320032     //
320033     if (count == 0)
320034     {
320035         c = 256;
320036     }
320037     else
320038     {
320039         c = count;
320040     }
320041     //
320042     // Read 'c' sectors.
320043     //
320044     for (; c > 0; c--)
320045     {
320046         s = ata_drq (drive, ATA_TIMEOUT);
320047         if (s < 0)
320048         {
320049             errset (errno);
320050             return (-1);
320051         }
320052         //
320053         // Write sector.
320054         //
320055         for (i = 0; i < (ATA_SECTOR_SIZE / 2); i++)
320056         {
320057             out_16 (ata_table[drive].r_data, source[i]);
320058         }
320059     }
```

```
320060 //
320061 // Wait that the device returns ready.
320062 //
320063 s = ata_rdy (drive, ATA_TIMEOUT);
320064 if (s < 0)
320065 {
320066     errset (errno);
320067     return (-1);
320068 }
320069 //
320070 // Now flush cache.
320071 //
320072 out_8 (ata_table[drive].r_command,
320073        ATA_COMMAND_FLUSH_CACHE);
320074 //
320075 // Then wait that the device returns ready.
320076 //
320077 s = ata_rdy (drive, ATA_TIMEOUT_FLUSH);
320078 if (s < 0)
320079 {
320080     errset (errno);
320081     return (-1);
320082 }
320083 //
320084 // Return.
320085 //
320086 return (0);
320087 }
```

94.4.7 kernel/driver/ata/ata_device.c

Si veda la sezione [93.2](#).

```
330001 #include <kernel/driver/ata.h>
330002 #include <kernel/lib_s.h>
330003 #include <kernel/ibm_i386.h>
330004 #include <stdint.h>
```

```
330005 #include <errno.h>
330006 //-----
330007 int
330008 ata_device (int drive, unsigned int sector)
330009 {
330010     unsigned char device;
330011     int s;
330012     //
330013     // Verify 'drive' argument.
330014     //
330015     s = ata_valid (drive);
330016     if (s < 0)
330017     {
330018         errset (EINVAL);
330019         return (-1);
330020     }
330021     //
330022     // Start building the 'device' register.
330023     //
330024     device = 0;
330025     //
330026     // The access will be LBA: no CHS at all here!
330027     //
330028     device |= ATA_DEVICE_LBA;
330029     //
330030     // Set the device number, relative to the bus.
330031     //
330032     device |= ata_table[drive].target;
330033     //
330034     // Put the highest four bits of the sector number,
330035     // that can use at most 28 bits.
330036     //
330037     device |= ((sector & 0x0F000000) >> 24);
330038     //
330039     // Must select the new drive.
330040     //
330041     out_8 (ata_table[drive].r_device, device);
```



```
330042 //
330043 // Wait for selected drive ready.
330044 //
330045 s = ata_rdy (drive, ATA_TIMEOUT);
330046 if (s < 0)
330047 {
330048     errset (errno);
330049     return (-1);
330050 }
330051 //
330052 // Ok.
330053 //
330054 return (0);
330055 }
```

94.4.8 kernel/driver/ata/ata_drq.c

Si veda la sezione [93.2](#).

```
340001 #include <kernel/driver/ata.h>
340002 #include <kernel/lib_s.h>
340003 #include <kernel/ibm_i386.h>
340004 #include <stdint.h>
340005 #include <errno.h>
340006 //-----
340007 int
340008 ata_drq (int drive, clock_t timeout)
340009 {
340010     clock_t time_start;
340011     clock_t time_now;
340012     clock_t time_elapsed;
340013     int status;
340014     //
340015     // The timeout value must be at least two.
340016     //
340017     if (timeout < 2)
340018     {
```

```
340019     timeout = 2;
340020     }
340021     //
340022     // Get the status register.
340023     //
340024     time_elapsed = 0;
340025     time_start = s_clock ((pid_t) 0);
340026     while (time_elapsed < timeout)
340027     {
340028         time_now = s_clock ((pid_t) 0);
340029         time_elapsed = time_now - time_start;
340030         //
340031         status = in_8 (ata_table[drive].r_status);
340032         //
340033         // Is it BSY?
340034         //
340035         if (status & ATA_STATUS_BSY)
340036         {
340037             //
340038             // Read the status again.
340039             //
340040             continue;
340041         }
340042         //
340043         // No more busy, but check for errors.
340044         //
340045         if (status & ATA_STATUS_DF)
340046         {
340047             k_printf ("%s] ERROR: drive %i fault\n",
340048                 __func__, drive);
340049             ata_reset (drive);
340050             errset (E_HARDWARE_FAULT);
340051             return (-1);
340052         }
340053         //
340054         if (status & ATA_STATUS_ERR)
340055         {
```

```

340056         k_printf ("%s] ERROR: drive %i error\n",
340057                 __func__, drive);
340058         ata_reset (drive);
340059         errset (E_DRIVER_FAULT);
340060         return (-1);
340061     }
340062     //
340063     // Now check for the DRQ.
340064     //
340065     if (status & ATA_STATUS_DRQ)
340066     {
340067         //
340068         // Ok.
340069         //
340070         return (0);
340071     }
340072 }
340073 //
340074 // Sorry: time out!
340075 //
340076 k_printf ("%s] ERROR: drive %i timeout\n", __func__,
340077         drive);
340078 errset (ETIME);
340079 return (-1);
340080 }

```

94.4.9 kernel/driver/ata/ata_init.c

Si veda la sezione [93.2](#).

```

350001 #include <kernel/driver/ata.h>
350002 #include <kernel/lib_k.h>
350003 #include <kernel/ibm_i386.h>
350004 #include <stdint.h>
350005 #include <errno.h>
350006 //-----
350007 void

```

```
350008 ata_init (void)
350009 {
350010     unsigned char status;
350011     int s;
350012     int d;
350013     //
350014     // Set bus numbers and I/O ports for each possible
350015     // drive.
350016     // I/O ports are related to the bus, so every couple
350017     // of
350018     // drive has the same ports.
350019     //
350020     if (ATA_MAX_DEVICES > 0)
350021     {
350022         ata_table[0].bus = 0;
350023         ata_table[0].r_data = ATA0_DATA;
350024         ata_table[0].r_feature = ATA0_FEATURE;
350025         ata_table[0].r_error = ATA0_ERROR;
350026         ata_table[0].r_count = ATA0_COUNT;
350027         ata_table[0].r_low = ATA0_LOW;
350028         ata_table[0].r_mid = ATA0_MID;
350029         ata_table[0].r_high = ATA0_HIGH;
350030         ata_table[0].r_device = ATA0_DEVICE;
350031         ata_table[0].r_command = ATA0_COMMAND;
350032         ata_table[0].r_status = ATA0_STATUS;
350033         ata_table[0].r_control = ATA0_CONTROL;
350034         ata_table[0].r_alternate = ATA0_ALTERNATE;
350035         ata_table[0].target = ATA_DEVICE_MASTER;
350036         //
350037         ata_table[1].bus = 0;
350038         ata_table[1].r_data = ATA0_DATA;
350039         ata_table[1].r_feature = ATA0_FEATURE;
350040         ata_table[1].r_error = ATA0_ERROR;
350041         ata_table[1].r_count = ATA0_COUNT;
350042         ata_table[1].r_low = ATA0_LOW;
350043         ata_table[1].r_mid = ATA0_MID;
350044         ata_table[1].r_high = ATA0_HIGH;
```

```
350045     ata_table[1].r_device = ATA0_DEVICE;
350046     ata_table[1].r_command = ATA0_COMMAND;
350047     ata_table[1].r_status = ATA0_STATUS;
350048     ata_table[1].r_control = ATA0_CONTROL;
350049     ata_table[1].r_alternate = ATA0_ALTERNATE;
350050     ata_table[1].target = ATA_DEVICE_SLAVE;
350051     }
350052     //
350053     if (ATA_MAX_DEVICES > 2)
350054     {
350055         ata_table[2].bus = 1;
350056         ata_table[2].r_data = ATA1_DATA;
350057         ata_table[2].r_feature = ATA1_FEATURE;
350058         ata_table[2].r_error = ATA1_ERROR;
350059         ata_table[2].r_count = ATA1_COUNT;
350060         ata_table[2].r_low = ATA1_LOW;
350061         ata_table[2].r_mid = ATA1_MID;
350062         ata_table[2].r_high = ATA1_HIGH;
350063         ata_table[2].r_device = ATA1_DEVICE;
350064         ata_table[2].r_command = ATA1_COMMAND;
350065         ata_table[2].r_status = ATA1_STATUS;
350066         ata_table[2].r_control = ATA1_CONTROL;
350067         ata_table[2].r_alternate = ATA1_ALTERNATE;
350068         ata_table[2].target = ATA_DEVICE_MASTER;
350069         //
350070         ata_table[3].bus = 1;
350071         ata_table[3].r_data = ATA1_DATA;
350072         ata_table[3].r_feature = ATA1_FEATURE;
350073         ata_table[3].r_error = ATA1_ERROR;
350074         ata_table[3].r_count = ATA1_COUNT;
350075         ata_table[3].r_low = ATA1_LOW;
350076         ata_table[3].r_mid = ATA1_MID;
350077         ata_table[3].r_high = ATA1_HIGH;
350078         ata_table[3].r_device = ATA1_DEVICE;
350079         ata_table[3].r_command = ATA1_COMMAND;
350080         ata_table[3].r_status = ATA1_STATUS;
350081         ata_table[3].r_control = ATA1_CONTROL;
```

```
350082     ata_table[3].r_alternate = ATA1_ALTERNATE;
350083     ata_table[3].target = ATA_DEVICE_SLAVE;
350084 }
350085 //
350086 if (ATA_MAX_DEVICES > 4)
350087 {
350088     ata_table[4].bus = 2;
350089     ata_table[4].r_data = ATA2_DATA;
350090     ata_table[4].r_feature = ATA2_FEATURE;
350091     ata_table[4].r_error = ATA2_ERROR;
350092     ata_table[4].r_count = ATA2_COUNT;
350093     ata_table[4].r_low = ATA2_LOW;
350094     ata_table[4].r_mid = ATA2_MID;
350095     ata_table[4].r_high = ATA2_HIGH;
350096     ata_table[4].r_device = ATA2_DEVICE;
350097     ata_table[4].r_command = ATA2_COMMAND;
350098     ata_table[4].r_status = ATA2_STATUS;
350099     ata_table[4].r_control = ATA2_CONTROL;
350100     ata_table[4].r_alternate = ATA2_ALTERNATE;
350101     ata_table[4].target = ATA_DEVICE_MASTER;
350102 //
350103     ata_table[5].bus = 2;
350104     ata_table[5].r_data = ATA2_DATA;
350105     ata_table[5].r_feature = ATA2_FEATURE;
350106     ata_table[5].r_error = ATA2_ERROR;
350107     ata_table[5].r_count = ATA2_COUNT;
350108     ata_table[5].r_low = ATA2_LOW;
350109     ata_table[5].r_mid = ATA2_MID;
350110     ata_table[5].r_high = ATA2_HIGH;
350111     ata_table[5].r_device = ATA2_DEVICE;
350112     ata_table[5].r_command = ATA2_COMMAND;
350113     ata_table[5].r_status = ATA2_STATUS;
350114     ata_table[5].r_control = ATA2_CONTROL;
350115     ata_table[5].r_alternate = ATA2_ALTERNATE;
350116     ata_table[5].target = ATA_DEVICE_SLAVE;
350117 }
350118 //
```

```
350119     if (ATA_MAX_DEVICES > 6)
350120     {
350121         ata_table[6].bus = 3;
350122         ata_table[6].r_data = ATA3_DATA;
350123         ata_table[6].r_feature = ATA3_FEATURE;
350124         ata_table[6].r_error = ATA3_ERROR;
350125         ata_table[6].r_count = ATA3_COUNT;
350126         ata_table[6].r_low = ATA3_LOW;
350127         ata_table[6].r_mid = ATA3_MID;
350128         ata_table[6].r_high = ATA3_HIGH;
350129         ata_table[6].r_device = ATA3_DEVICE;
350130         ata_table[6].r_command = ATA3_COMMAND;
350131         ata_table[6].r_status = ATA3_STATUS;
350132         ata_table[6].r_control = ATA3_CONTROL;
350133         ata_table[6].r_alternate = ATA3_ALTERNATE;
350134         ata_table[6].target = ATA_DEVICE_MASTER;
350135         //
350136         ata_table[7].bus = 3;
350137         ata_table[7].r_data = ATA3_DATA;
350138         ata_table[7].r_feature = ATA3_FEATURE;
350139         ata_table[7].r_error = ATA3_ERROR;
350140         ata_table[7].r_count = ATA3_COUNT;
350141         ata_table[7].r_low = ATA3_LOW;
350142         ata_table[7].r_mid = ATA3_MID;
350143         ata_table[7].r_high = ATA3_HIGH;
350144         ata_table[7].r_device = ATA3_DEVICE;
350145         ata_table[7].r_command = ATA3_COMMAND;
350146         ata_table[7].r_status = ATA3_STATUS;
350147         ata_table[7].r_control = ATA3_CONTROL;
350148         ata_table[7].r_alternate = ATA3_ALTERNATE;
350149         ata_table[7].target = ATA_DEVICE_SLAVE;
350150     }
350151     //
350152     // Scan and eliminate buses with no drive at all.
350153     //
350154     for (d = 0; d < ATA_MAX_DEVICES; d += 2)
350155     {
```

```
350156     status = in_8 (ata_table[d].r_status);
350157     if (status == 0xFF)
350158     {
350159         ata_table[d].present = 0;
350160         ata_table[d + 1].present = 0;
350161     }
350162     else
350163     {
350164         ata_table[d].present = 1;
350165         ata_table[d + 1].present = 1;
350166     }
350167 }
350168 //
350169 // Identify drives.
350170 //
350171 for (d = 0; d < ATA_MAX_DEVICES; d++)
350172 {
350173     if (ata_table[d].present == 0)
350174     {
350175         continue;
350176     }
350177 //
350178 // Register 'device'.
350179 //
350180 s = ata_device (d, 0);
350181 if (s < 0)
350182 {
350183     errset (errno);
350184     k_perror (NULL);
350185     ata_table[d].present = 0;
350186     continue;
350187 }
350188 //
350189 // Send command 'IDENTIFY DEVICE'
350190 //
350191 s =
350192     ata_cmd_identify_device (d, &(ata_table[d].id[0]));
```



```
350193     if (s < 0)
350194     {
350195         ata_table[d].present = 0;
350196         continue;
350197     }
350198     //
350199     // Verify again if the drive is present: the
350200     // function might have found that it does not
350201     // exist.
350202     //
350203     if (ata_table[d].present == 0)
350204     {
350205         continue;
350206     }
350207     //
350208     // Find total sectors (for 28 bit LBA access).
350209     // It is written
350210     // inside the integer formed by 'identity[60]'
350211     // and
350212     // 'identity[61]', considering it in little
350213     // endian mode.
350214     // It is taken the pointer to 'identity[60]',
350215     // transformed
350216     // into a pointer to a 32 bit integer, and then
350217     // dereferenced
350218     // again.
350219     //
350220     ata_table[d].sectors
350221     = *((uint32_t *) & (ata_table[d].id[60]));
350222     //
350223     // Check if the size value is right.
350224     //
350225     if (ata_table[d].sectors == 0)
350226     {
350227         ata_table[d].present = 0;
350228     }
350229     else
```

```
350230     {
350231         //
350232         // Show info.
350233         //
350234         k_printf ("%s] ATA drive %i size %i Kib\n",
350235                 __func__, d, ata_table[d].sectors / 2);
350236     }
350237 }
350238 }
```

94.4.10 kernel/driver/ata/ata_lba28.c



Si veda la sezione [93.2](#).

```
360001 #include <kernel/driver/ata.h>
360002 #include <kernel/lib_s.h>
360003 #include <kernel/ibm_i386.h>
360004 #include <stdint.h>
360005 #include <errno.h>
360006 //-----
360007 int
360008 ata_lba28 (int drive, unsigned int sector,
360009           unsigned char count)
360010 {
360011     int s;
360012     unsigned char low;
360013     unsigned char mid;
360014     unsigned char high;
360015     //
360016     // Register 'device'.
360017     //
360018     s = ata_device (drive, sector);
360019     if (s < 0)
360020     {
360021         k_perror (NULL);
360022
360023         errset (errno);
```

```
360024     return (-1);
360025     }
360026     //
360027     // Register 'control', to set nIEN.
360028     //
360029     out_8 (ata_table[drive].r_control, ATA_CONTROL_NIEN);
360030     //
360031     // Register 'feature'. not used.
360032     //
360033     out_8 (ata_table[drive].r_feature, 0);
360034     //
360035     // Register 'count'
360036     //
360037     out_8 (ata_table[drive].r_count, count);
360038     //
360039     // Registers 'low', 'mid', 'high'.
360040     //
360041     low = (sector & 0x000000FF);
360042     mid = ((sector & 0x0000FF00) >> 8);
360043     high = ((sector & 0x00FF0000) >> 16);
360044     //
360045     out_8 (ata_table[drive].r_low, low);
360046     out_8 (ata_table[drive].r_mid, mid);
360047     out_8 (ata_table[drive].r_high, high);
360048     //
360049     // Ok.
360050     //
360051     return (0);
360052 }
```

94.4.11 kernel/driver/ata/ata_public.c

Si veda la sezione [93.2](#).

```
370001 #include <kernel/driver/ata.h>
370002 //-----
370003 ata_t ata_table[ATA_MAX_DEVICES];
```

370004

//-----

94.4.12 kernel/driver/ata/ata_rdy.c

<<

Si veda la sezione [93.2](#).

```
380001 #include <kernel/driver/ata.h>
380002 #include <kernel/lib_s.h>
380003 #include <kernel/ibm_i386.h>
380004 #include <stdint.h>
380005 #include <errno.h>
380006 //-----
380007 int
380008 ata_rdy (int drive, clock_t timeout)
380009 {
380010     clock_t time_start;
380011     clock_t time_now;
380012     clock_t time_elapsed;
380013     unsigned char status;
380014     //
380015     // The timeout value must be at least two.
380016     //
380017     if (timeout < 2)
380018     {
380019         timeout = 2;
380020     }
380021     //
380022     // Get the status register.
380023     //
380024     time_elapsed = 0;
380025     time_start = s_clock ((pid_t) 0);
380026     while (time_elapsed < timeout)
380027     {
380028         time_now = s_clock ((pid_t) 0);
380029         time_elapsed = time_now - time_start;
380030         //
380031         status = in_8 (ata_table[drive].r_status);
```

```
380032 //
380033 // Is it BSY?
380034 //
380035 if (status & ATA_STATUS_BSY)
380036 {
380037 //
380038 // Read the status again.
380039 //
380040 continue;
380041 }
380042 //
380043 // No more busy, but check for errors.
380044 //
380045 if (status & ATA_STATUS_DF)
380046 {
380047 k_printf ("%s] ERROR: drive %i fault\n",
380048           __func__, drive);
380049 ata_reset (drive);
380050 errset (E_HARDWARE_FAULT);
380051 return (-1);
380052 }
380053 //
380054 if (status & ATA_STATUS_ERR)
380055 {
380056 k_printf ("%s] ERROR: drive %i error\n",
380057           __func__, drive);
380058 ata_reset (drive);
380059 errset (E_DRIVER_FAULT);
380060 return (-1);
380061 }
380062 //
380063 // Otherwise: ok.
380064 //
380065 return (0);
380066 }
380067 //
380068 // Sorry: time out!
```

```
380069 //
380070 k_printf ("%s] ERROR: drive %i timeout\n", __func__,
380071           drive);
380072 errset (ETIME);
380073 return (-1);
380074 }
```

94.4.13 kernel/driver/ata/ata_reset.c

<<

Si veda la sezione [93.2](#).

```
390001 #include <kernel/driver/ata.h>
390002 #include <kernel/lib_s.h>
390003 #include <kernel/ibm_i386.h>
390004 #include <stdint.h>
390005 #include <errno.h>
390006 //-----
390007 void
390008 ata_reset (int drive)
390009 {
390010     out_8 (ata_table[drive].r_control, ATA_CONTROL_SRST);
390011     out_8 (ata_table[drive].r_control, 0);
390012 }
```

94.4.14 kernel/driver/ata/ata_valid.c

<<

Si veda la sezione [93.2](#).

```
400001 #include <kernel/driver/ata.h>
400002 #include <errno.h>
400003 //-----
400004 int
400005 ata_valid (int drive)
400006 {
400007     //
400008     // Verify if 'drive' argument is possible.
400009     //
```

```
400010     if (drive < 0 || drive >= ATA_MAX_DEVICES)
400011     {
400012         errset (EINVAL);
400013         return (-1);
400014     }
400015     //
400016     // Verify if 'drive' is present at the moment.
400017     //
400018     if (ata_table[drive].present == 0)
400019     {
400020         errset (E_NO_MEDIUM);
400021         return (-1);
400022     }
400023     //
400024     // OK.
400025     //
400026     return (0);
400027 }
```

94.4.15 kernel/driver/kbd.h

Si veda la sezione [93.10](#).

```
410001 #ifndef _KERNEL_DRIVER_KBD_H
410002 #define _KERNEL_DRIVER_KBD_H      1
410003 //-----
410004 #include <stdbool.h>
410005 //-----
410006 //
410007 // Keyboard status.
410008 //
410009 typedef struct
410010 {
410011     bool shift;
410012     bool shift_lock;
410013     bool ctrl;
410014     bool alt;
```

```
410015     bool echo;
410016     unsigned char key;
410017     unsigned char map1[128];
410018     unsigned char map2[128];
410019 } kbd_t;
410020 //
410021 extern kbd_t kbd;
410022 //-----
410023 void kbd_load (void);
410024 void kbd_isr (void);
410025 //-----
410026 #endif
```

94.4.16 kernel/driver/kbd/kbd_isr.c

«

Si veda la sezione [93.10](#).

```
420001 #include <kernel/driver/kbd.h>
420002 #include <kernel/ibm_i386.h>
420003 //-----
420004 void
420005 kbd_isr (void)
420006 {
420007     //
420008     // Get a character from the keyboard channel.
420009     //
420010     unsigned char scancode = (int) in_8 (0x60);
420011     //
420012     // Check for [Shift], [Shift-Lock], [Ctrl] and [Alt]
420013     // keys.
420014     //
420015     switch (scancode)
420016     {
420017         case 0x2A:
420018             kbd.shift = 1;
420019             break;
420020         case 0x36:
```



```
420021     kbd.shift = 1;
420022     break;
420023     case 0xAA:
420024         kbd.shift = 0;
420025         break;
420026     case 0xB6:
420027         kbd.shift = 0;
420028         break;
420029     case 0x1D:
420030         kbd.ctrl = 1;
420031         break;
420032     case 0x9D:
420033         kbd.ctrl = 0;
420034         break;
420035     case 0x38:
420036         kbd.alt = 1;
420037         break;
420038     case 0xB8:
420039         kbd.alt = 0;
420040         break;
420041     case 0x3A:
420042         kbd.shift_lock = !kbd.shift_lock;
420043         break;
420044     }
420045     //
420046     // Check for usual keys, but only if 'kbd.key' is
420047     // currently empty.
420048     //
420049     if (scancode <= 127 && kbd.ctrl && kbd.key == 0)
420050     {
420051         //
420052         // Something was pressed, combined with [Ctrl].
420053         //
420054         switch (kbd.map1[scancode])
420055         {
420056             case 'a':
420057                 kbd.key = 0x01;
```

```
420058         break;           // SOH
420059     case 'b':
420060         kbd.key = 0x02;
420061         break;           // STX
420062     case 'c':
420063         kbd.key = 0x03;
420064         break;           // ETX
420065     case 'd':
420066         kbd.key = 0x04;
420067         break;           // EOT
420068     case 'e':
420069         kbd.key = 0x05;
420070         break;           // ENQ
420071     case 'f':
420072         kbd.key = 0x06;
420073         break;           // ACK
420074     case 'g':
420075         kbd.key = 0x07;
420076         break;           // BEL
420077     case 'h':
420078         kbd.key = 0x08;
420079         break;           // BS
420080     case 'i':
420081         kbd.key = 0x09;
420082         break;           // HT
420083     case 'j':
420084         kbd.key = 0x0A;
420085         break;           // LF
420086     case 'k':
420087         kbd.key = 0x0B;
420088         break;           // VT
420089     case 'l':
420090         kbd.key = 0x0C;
420091         break;           // FF
420092     case 'm':
420093         kbd.key = 0x0D;
420094         break;           // CR
```

```
420095     case 'n':
420096         kbd.key = 0x0E;
420097         break;           // SO
420098     case 'o':
420099         kbd.key = 0x0F;
420100         break;           // SI
420101     case 'p':
420102         kbd.key = 0x10;
420103         break;           // DLE
420104     case 'q':
420105         kbd.key = 0x11;
420106         break;           // DC1
420107     case 'r':
420108         kbd.key = 0x12;
420109         break;           // DC2
420110     case 's':
420111         kbd.key = 0x13;
420112         break;           // DC3
420113     case 't':
420114         kbd.key = 0x14;
420115         break;           // DC4
420116     case 'u':
420117         kbd.key = 0x15;
420118         break;           // NAK
420119     case 'v':
420120         kbd.key = 0x16;
420121         break;           // SYN
420122     case 'w':
420123         kbd.key = 0x17;
420124         break;           // ETB
420125     case 'x':
420126         kbd.key = 0x18;
420127         break;           // CAN
420128     case 'y':
420129         kbd.key = 0x19;
420130         break;           // EM
420131     case 'z':
```

```
420132     kbd.key = 0x1A;
420133     break;          // SUB
420134     case '[':
420135         kbd.key = 0x1B;
420136         break;          // ESC
420137     case '\\':
420138         kbd.key = 0x1C;
420139         break;          // FS
420140     case ']':
420141         kbd.key = 0x1D;
420142         break;          // GS
420143     case '^':
420144         kbd.key = 0x1E;
420145         break;          // RS
420146     case '_':
420147         kbd.key = 0x1F;
420148         break;          // US
420149     }
420150 }
420151 else if (scancode <= 127 && kbd.key == 0
420152         && kbd.map1[scancode] != 0)
420153     {
420154         //
420155         // Something was pressed, but no [Ctrl] is
420156         // there.
420157         //
420158         if (kbd.shift || kbd.shift_lock)
420159             {
420160                 kbd.key = kbd.map2[scancode];
420161             }
420162         else
420163             {
420164                 kbd.key = kbd.map1[scancode];
420165             }
420166     }
420167 }
```

94.4.17 kernel/driver/kbd/kbd_load.c



Si veda la sezione [93.10](#).

```
430001 #include <kernel/driver/kbd.h>
430002 //-----
430003 void
430004 kbd_load (void)
430005 {
430006     int i;
430007     //
430008     // Reset the map.
430009     //
430010     for (i = 0; i <= 127; i++)
430011     {
430012         kbd.map1[i] = 0;
430013         kbd.map2[i] = 0;
430014     }
430015     //
430016     // Association for an italian map, but only with
430017     // ASCII characters
430018     // (there are no accented letters).
430019     //
430020     kbd.map1[1] = 27;
430021     kbd.map2[1] = 27;           // ESC
430022     kbd.map1[2] = '1';
430023     kbd.map2[2] = '!';
430024     kbd.map1[3] = '2';
430025     kbd.map2[3] = '"';
430026     kbd.map1[4] = '3';
430027     kbd.map2[4] = 'L';       // 3, £
430028     kbd.map1[5] = '4';
430029     kbd.map2[5] = '$';
430030     kbd.map1[6] = '5';
430031     kbd.map2[6] = '%';
430032     kbd.map1[7] = '6';
430033     kbd.map2[7] = '&';
430034     kbd.map1[8] = '7';
```

```
430035 kbd.map2[8] = '/';
430036 kbd.map1[9] = '8';
430037 kbd.map2[9] = '(';
430038 kbd.map1[10] = '9';
430039 kbd.map2[10] = ')';
430040 kbd.map1[11] = '0';
430041 kbd.map2[11] = '=';
430042 kbd.map1[12] = '\\';
430043 kbd.map2[12] = '?';
430044 kbd.map1[13] = 'i';
430045 kbd.map2[13] = '^'; // ì, ^
430046 kbd.map1[14] = '\\b';
430047 kbd.map2[14] = '\\b'; // Backspace
430048 kbd.map1[15] = '\\t';
430049 kbd.map2[15] = '\\t';
430050 kbd.map1[16] = 'q';
430051 kbd.map2[16] = 'Q';
430052 kbd.map1[17] = 'w';
430053 kbd.map2[17] = 'W';
430054 kbd.map1[18] = 'e';
430055 kbd.map2[18] = 'E';
430056 kbd.map1[19] = 'r';
430057 kbd.map2[19] = 'R';
430058 kbd.map1[20] = 't';
430059 kbd.map2[20] = 'T';
430060 kbd.map1[21] = 'y';
430061 kbd.map2[21] = 'Y';
430062 kbd.map1[22] = 'u';
430063 kbd.map2[22] = 'U';
430064 kbd.map1[23] = 'i';
430065 kbd.map2[23] = 'I';
430066 kbd.map1[24] = 'o';
430067 kbd.map2[24] = 'O';
430068 kbd.map1[25] = 'p';
430069 kbd.map2[25] = 'P';
430070 kbd.map1[26] = '[';
430071 kbd.map2[26] = '{'; // è, é
```

```
430072 kbd.map1[27] = ']' ;
430073 kbd.map2[27] = '}' ; // +, *
430074 kbd.map1[28] = '\n' ;
430075 kbd.map2[28] = '\n' ; // Enter
430076 kbd.map1[30] = 'a' ;
430077 kbd.map2[30] = 'A' ;
430078 kbd.map1[31] = 's' ;
430079 kbd.map2[31] = 'S' ;
430080 kbd.map1[32] = 'd' ;
430081 kbd.map2[32] = 'D' ;
430082 kbd.map1[33] = 'f' ;
430083 kbd.map2[33] = 'F' ;
430084 kbd.map1[34] = 'g' ;
430085 kbd.map2[34] = 'G' ;
430086 kbd.map1[35] = 'h' ;
430087 kbd.map2[35] = 'H' ;
430088 kbd.map1[36] = 'j' ;
430089 kbd.map2[36] = 'J' ;
430090 kbd.map1[37] = 'k' ;
430091 kbd.map2[37] = 'K' ;
430092 kbd.map1[38] = 'l' ;
430093 kbd.map2[38] = 'L' ;
430094 kbd.map1[39] = '@' ;
430095 kbd.map2[39] = '@' ; // ò, ç
430096 kbd.map1[40] = '#' ;
430097 kbd.map2[40] = '#' ; // à, °
430098 kbd.map1[41] = '\\ ' ;
430099 kbd.map2[41] = '| ' ;
430100 kbd.map1[43] = 'u' ;
430101 kbd.map2[43] = 'U' ; // ù, §
430102 kbd.map1[44] = 'z' ;
430103 kbd.map2[44] = 'Z' ;
430104 kbd.map1[45] = 'x' ;
430105 kbd.map2[45] = 'X' ;
430106 kbd.map1[46] = 'c' ;
430107 kbd.map2[46] = 'C' ;
430108 kbd.map1[47] = 'v' ;
```

```
430109 kbd.map2[47] = 'V';
430110 kbd.map1[48] = 'b';
430111 kbd.map2[48] = 'B';
430112 kbd.map1[49] = 'n';
430113 kbd.map2[49] = 'N';
430114 kbd.map1[50] = 'm';
430115 kbd.map2[50] = 'M';
430116 kbd.map1[51] = ',';
430117 kbd.map2[51] = ';';
430118 kbd.map1[52] = '.';
430119 kbd.map2[52] = ':';
430120 kbd.map1[53] = '-';
430121 kbd.map2[53] = '_';
430122 kbd.map1[56] = '<';
430123 kbd.map2[56] = '>';
430124 kbd.map1[57] = ' ';
430125 kbd.map2[57] = ' ';
430126 //
430127 kbd.map1[55] = '*';
430128 kbd.map2[55] = '*';
430129 kbd.map1[71] = '7';
430130 kbd.map2[71] = '7';
430131 kbd.map1[72] = '8';
430132 kbd.map2[72] = '8';
430133 kbd.map1[73] = '9';
430134 kbd.map2[73] = '9';
430135 kbd.map1[74] = '-';
430136 kbd.map2[74] = '-';
430137 kbd.map1[75] = '4';
430138 kbd.map2[75] = '4';
430139 kbd.map1[76] = '5';
430140 kbd.map2[76] = '5';
430141 kbd.map1[77] = '6';
430142 kbd.map2[77] = '6';
430143 kbd.map1[78] = '+';
430144 kbd.map2[78] = '+';
430145 kbd.map1[79] = '1';
```



```

430146     kbd.map2[79] = '1';
430147     kbd.map1[80] = '2';
430148     kbd.map2[80] = '2';
430149     kbd.map1[81] = '3';
430150     kbd.map2[81] = '3';
430151     kbd.map1[82] = '0';
430152     kbd.map2[82] = '0';
430153     kbd.map1[83] = '.';
430154     kbd.map2[83] = '.';
430155     kbd.map1[92] = '/';
430156     kbd.map2[92] = '/';
430157     kbd.map1[96] = '\n';
430158     kbd.map2[96] = '\n'; // Enter
430159 }

```

94.4.18 kernel/driver/kbd/kbd_public.c

Si veda la sezione [93.10](#).

```

440001 #include <kernel/driver/kbd.h>
440002 //-----
440003 kbd_t kbd;

```

94.4.19 kernel/driver/nic/ne2k.h

Si veda la sezione [93.16](#).

```

450001 #ifndef _KERNEL_DRIVER_NIC_NE2K_H
450002 #define _KERNEL_DRIVER_NIC_NE2K_H    1
450003 //-----
450004 #include <stdint.h>
450005 #include <sys/types.h>
450006 #include <unistd.h>
450007 #include <time.h>
450008 //-----
450009 //
450010 // Command register, available on all pages.

```

```
450011 //
450012 #define NE2K_CR          0x00    // rw
450013 //
450014 // Page 0 registers
450015 //
450016 #define NE2K_CLDA0       0x01    // r-
450017 #define NE2K_PSTART     0x01    // -w
450018 #define NE2K_CLDA1       0x02    // r-
450019 #define NE2K_PSTOP      0x02    // -w
450020 #define NE2K_BNRY       0x03    // rw
450021 #define NE2K_TSR        0x04    // r-
450022 #define NE2K_TPSR       0x04    // -w
450023 #define NE2K_NCR        0x05    // r-
450024 #define NE2K_TBCR0      0x05    // -w
450025 #define NE2K_FIFO       0x06    // r-
450026 #define NE2K_TBCR1      0x06    // -w
450027 #define NE2K_ISR        0x07    // rw
450028 #define NE2K_CRDA0      0x08    // r-
450029 #define NE2K_RSAR0      0x08    // -w
450030 #define NE2K_CRDA1      0x09    // r-
450031 #define NE2K_RSAR1      0x09    // -w
450032 #define NE2K_RBCR0      0x0A    // -w
450033 #define NE2K_RBCR1      0x0B    // -w
450034 #define NE2K_RSR        0x0C    // r-
450035 #define NE2K_RCR        0x0C    // -w
450036 #define NE2K_CNTR0      0x0D    // r-
450037 #define NE2K_TCR        0x0D    // -w
450038 #define NE2K_CNTR1      0x0E    // r-
450039 #define NE2K_DCR        0x0E    // -w
450040 #define NE2K_CNTR2      0x0F    // r-
450041 #define NE2K_IMR        0x0F    // -w
450042 //
450043 // Page 1 registers: all read and write
450044 //
450045 #define NE2K_PAR0       0x01    // rw
450046 #define NE2K_PAR1       0x02    // rw
450047 #define NE2K_PAR2       0x03    // rw
```

```
450048 #define NE2K_PAR3      0x04    // rw
450049 #define NE2K_PAR4      0x05    // rw
450050 #define NE2K_PAR5      0x06    // rw
450051 #define NE2K_CURR      0x07    // rw
450052 #define NE2K_MAR0      0x08    // rw
450053 #define NE2K_MAR1      0x09    // rw
450054 #define NE2K_MAR2      0x0A    // rw
450055 #define NE2K_MAR3      0x0B    // rw
450056 #define NE2K_MAR4      0x0C    // rw
450057 #define NE2K_MAR5      0x0D    // rw
450058 #define NE2K_MAR6      0x0E    // rw
450059 #define NE2K_MAR7      0x0F    // rw
450060 //
450061 // Page 2 registers: read version of the same registers
450062 // inside page 1:
450063 //
450064 //     NE2K_PSTART      0x01    // -w
450065 //     NE2K_PSTOP       0x02    // -w
450066 //     --               0x03
450067 //     NE2K_TPSR        0x04    // -w
450068 //     --               0x05
450069 //     --               0x06
450070 //     --               0x07
450071 //     --               0x08
450072 //     --               0x09
450073 //     --               0x0A
450074 //     --               0x0B
450075 //     NE2K_RCR         0x0C    // -w
450076 //     NE2K_TCR         0x0D    // -w
450077 //     NE2K_DCR         0x0E    // -w
450078 //     NE2K_IMR        0x0F    // -w
450079 //
450080 // Extra registers, outside pages.
450081 //
450082 #define NE2K_DATA      0x10    // Data port.
450083 #define NE2K_RESET    0x1f    // Reset port.
450084 //
```

```

450085 // DMA buffer structure.
450086 //
450087 // 0x0000 .------.
450088 //          |          |
450089 //          |          |
450090 //          |          |          ???
450091 //          |          |
450092 //          |          |
450093 // 0x4000 |-----|
450094 //          | transmit buffer ring [1] |
450095 // 0x4600 |-----|
450096 //          |          |
450097 //          | receive buffer ring |
450098 //          |          |
450099 // 0xC000 |-----|
450100 //          |          |
450101 //          |          |          ???
450102 //          |          |
450103 //          |          |
450104 //          \-----' 0xFFFF
450105 //
450106 // [1] 0x600 is equal to 1536, that is the max space
450107 //       that a packet can occupy inside the ring buffer.
450108 //       So, there is the place for a
450109 //       single transmission packet.
450110 //
450111 // Local DMA addresses:
450112 //
450113 #define NE2K_TX_START 0x40 // Means: 0x4000
450114 #define NE2K_TX_STOP 0x46 // Means: 0x4600
450115
450116 #define NE2K_RX_START 0x46 // Means: 0x4600
450117 #define NE2K_RX_STOP 0xC0 // Means: 0xC000
450118 //
450119 // Transmit buffer (remote DMA).
450120 //
450121 #define NE2K_TX_BUFFER NE2K_TX_START

```

```
450122 //
450123 // SA-PROM (Station Address PROM) size
450124 //
450125 #define NE2K_SAPROM_SIZE 32 // 32 bytes
450126 //-----
450127 int ne2k_check (uintptr_t io);
450128 int ne2k_isr (uintptr_t io);
450129 int ne2k_isr_expect (uintptr_t io, unsigned int isr_expect);
450130 int ne2k_reset (uintptr_t io, void *address);
450131 int ne2k_rx (uintptr_t io);
450132 int ne2k_rx_reset (uintptr_t io);
450133 int ne2k_tx (uintptr_t io, void *buffer, size_t size);
450134 //
450135 void ne2k_test2 (int eth);
450136
450137
450138 //-----
450139 #endif
```

94.4.20 kernel/driver/nic/ne2k/ne2k_check.c



Si veda la sezione [93.16](#).

```
460001 #include <kernel/driver/nic/ne2k.h>
460002 #include <kernel/ibm_i386.h>
460003 #include <errno.h>
460004 //-----
460005 int
460006 ne2k_check (uintptr_t io)
460007 {
460008     int status;
460009     int reg_00;
460010     int reg_0d;
460011     //
460012     // Read and save the command register (CR, 0x00): if
460013     // it is really the
460014     // command register, should not be equal to 0xFF.
```

```

460015 //
460016 reg_00 = in_8 (io + NE2K_CR);
460017 if (reg_00 == 0xFF)
460018 {
460019     errset (E_DRIVER_FAULT);
460020     return (-1);
460021 }
460022 //
460023 // Command register (CR)
460024 // .------.
460025 // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
460026 // |-----|
460027 // | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0x61
460028 // `-----'
460029 // \___/ \_____/ |
460030 // | | STOP
460031 // | Abort/complete
460032 // | remote DMA
460033 // Register
460034 // page 1
460035 //
460036 // Stop and select page 1.
460037 //
460038 out_8 ((io + NE2K_CR), 0x61);
460039 //
460040 // Read, save and overwrite the register MAR5 (0x0D
460041 // at
460042 // page 1).
460043 //
460044 reg_0d = in_8 (io + NE2K_MAR5);
460045 out_8 (io + NE2K_MAR5, 0xFF);
460046 //
460047 // Command register (CR)
460048 // .------.
460049 // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
460050 // |-----|
460051 // | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21

```

```
460052 // '-----'
460053 // \_/_/ \_/_/_/ |
460054 // | | STOP
460055 // | Abort/complete
460056 // | remote DMA
460057 // Register
460058 // page 0
460059 //
460060 // Stop and select page 0.
460061 //
460062 out_8 ((io + NE2K_CR), 0x21);
460063 //
460064 // Read the tally counter 0 (CNTR0) to clear it.
460065 //
460066 in_8 (io + NE2K_CNTR0);
460067 //
460068 // Now the tally counter 0 (CNTR0) should be zero.
460069 //
460070 status = in_8 (io + NE2K_CNTR0);
460071 if (status)
460072 {
460073 //
460074 // The value obtained is not zero, so it is not
460075 // a NE2000 nic and the page change had probably
460076 // no effect. So, restore the values found
460077 // inside
460078 // 0x00 and 0x0D, without trying to change page.
460079 //
460080 out_8 (io, reg_00);
460081 out_8 ((io + 0x0D), reg_0d);
460082 errset (E_DRIVER_FAULT);
460083 return (-1);
460084 }
460085 //
460086 // Everything is ok: it might be a NE2000 nic.
460087 //
460088 return (0);
```

460089	}
--------	---

94.4.21 kernel/driver/nic/ne2k/ne2k_isr.c

<<

Si veda la sezione [93.16](#).

```
470001 #include <kernel/driver/pci.h>
470002 #include <kernel/driver/nic/ne2k.h>
470003 #include <kernel/ibm_i386.h>
470004 #include <errno.h>
470005 #include <kernel/lib_k.h>
470006 #include <kernel/lib_s.h>
470007 //-----
470008 int
470009 ne2k_isr (uintptr_t io)
470010 {
470011     int isr;
470012     //
470013     // Get ISR (interrupt status register).
470014     //
470015     isr = in_8 (io + NE2K_ISR);
470016     //
470017     //
470018     //
470019     if (isr & 0x01)
470020     {
470021         //
470022         // Frame received.
470023         //
470024         out_8 (io + NE2K_ISR, 0x01);
470025         ne2k_rx (io);
470026     }
470027     if (isr & 0x04)
470028     {
470029         //
470030         // Frame received with errors.
470031         //
```



```
470032     out_8 (io + NE2K_ISR, 0x04);
470033 }
470034 if (isr & 0x02)
470035 {
470036     //
470037     // Frame sent correctly.
470038     //
470039     ;
470040 }
470041 if (isr & 0x08)
470042 {
470043     //
470044     // Frame sent with errors.
470045     //
470046     ;
470047 }
470048 if (isr & 0x10)
470049 {
470050     k_printf ("OVERWRITE\n");
470051     out_8 (io + NE2K_ISR, 0x10);
470052     //
470053     // I don't understand if it works: Bochs just
470054     // don't accept
470055     // frames if they can make an overflow.
470056     //
470057     ne2k_rx (io);
470058     ne2k_rx_reset (io);
470059 }
470060 if (isr & 0x20)
470061 {
470062     //
470063     // Counter overflow.
470064     //
470065     out_8 (io + NE2K_ISR, 0x20);
470066 }
470067 if (isr & 0x40)
470068 {
```

```
470069      //
470070      // Remote DMA complete.
470071      //
470072      ;
470073  }
470074  if (isr & 0x80)
470075  {
470076      //
470077      // Reset status.
470078      //
470079      ;
470080  }
470081  //
470082  // End.
470083  //
470084  return (0);
470085 }
```

94.4.22 kernel/driver/nic/ne2k/ne2k_isr_expect.c

<<

Si veda la sezione [93.16](#).

```
480001 #include <kernel/driver/nic/ne2k.h>
480002 #include <kernel/ibm_i386.h>
480003 #include <kernel/lib_k.h>
480004 #include <errno.h>
480005 #include <unistd.h>
480006 #include <time.h>
480007 //-----
480008 #define DEBUG 0
480009 //-----
480010 int
480011 ne2k_isr_expect (uintptr_t io, unsigned int isr_expect)
480012 {
480013     int retry = 5;
480014     int status;
480015     //
```

```
480016 //
480017 //
480018 for (; retry > 0; retry--)
480019 {
480020     status = in_8 (io + NE2K_ISR);
480021     //
480022     if (status & isr_expect)
480023     {
480024         //
480025         // Reset the bit found true and exit loop.
480026         //
480027         out_8 ((io + NE2K_ISR), isr_expect);
480028         return (0);
480029     }
480030 }
480031 //
480032 // If ISR is zero, we assume that it is ok.
480033 //
480034 if (status == 0)
480035 {
480036     if (DEBUG)
480037     {
480038         k_printf ("[isr=0x%02x expect=0x%02x]",
480039                 status, isr_expect);
480040         return (0);
480041     }
480042     return (0);
480043 }
480044 else
480045 {
480046     if (DEBUG)
480047     {
480048         //
480049         // It is not zero, but we prefer to let it
480050         // go...
480051         //
480052         k_printf ("[isr=0x%02x expect=0x%02x]",
```

```
480053         status, isr_expect);
480054     return (0);
480055     }
480056     else
480057     {
480058         errset (E_DRIVER_FAULT);
480059         return (-1);
480060     }
480061     }
480062 }
```

94.4.23 kernel/driver/nic/ne2k/ne2k_reset.c

<<

Si veda la sezione [93.16](#).

```
490001 #include <kernel/driver/pci.h>
490002 #include <kernel/driver/nic/ne2k.h>
490003 #include <kernel/ibm_i386.h>
490004 #include <errno.h>
490005 #include <kernel/lib_k.h>
490006 #include <kernel/lib_s.h>
490007 //-----
490008 int
490009 ne2k_reset (uintptr_t io, void *address)
490010 {
490011     int status;
490012     int i;
490013     uint8_t sa_prom[NE2K_SAPROM_SIZE];
490014     uint8_t par[6];
490015     //
490016     // -----
490017     // RESET
490018     // -----
490019     //
490020     status = in_8 (io + NE2K_RESET);
490021     out_8 ((io + NE2K_RESET), 0xFF);
490022     out_8 ((io + NE2K_RESET), status);
```

```

490023 //
490024 // Interrupt status register (ISR)
490025 // .------.
490026 // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
490027 // |-----|
490028 // | 1 | ? | ? | ? | ? | ? | ? | ? | 0x80
490029 // `-----'
490030 // |
490031 // Reset status
490032 //
490033 // Verify to have reset the NIC.
490034 //
490035 status = ne2k_isr_expect (io, 0x80);
490036 if (status)
490037     {
490038         errset (errno);
490039         return (-1);
490040     }
490041 //
490042 // Reset all ISR register flags.
490043 //
490044 out_8 ((io + NE2K_ISR), 0xFF);
490045 //
490046 // -----
490047 // GET ETHERNET ADDRESS FROM SA-PROM (Station
490048 // Address PROM)
490049 // -----
490050 //
490051 // Command register (CR)
490052 // .------.
490053 // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
490054 // |-----|
490055 // | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
490056 // `-----'
490057 // \_____/ \_____/ |
490058 // | | STOP
490059 // | |

```

```

490060 // | Abort/complete
490061 // | remote DMA
490062 // |
490063 // Register
490064 // page 0
490065 //
490066 out_8 ((io + NE2K_CR), 0x21);
490067 //
490068 // Interrupt status register (ISR)
490069 // .-----.
490070 // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
490071 // |-----|
490072 // | 1 | ? | ? | ? | ? | ? | ? | ? | 0x80
490073 // `-----'
490074 // |
490075 // Reset status
490076 //
490077 // Verify to have reset the NIC.
490078 //
490079 status = ne2k_isr_expect (io, 0x80);
490080 if (status)
490081 {
490082     errset (errno);
490083     return (-1);
490084 }
490085 //
490086 // Data configuration register (DCR)
490087 // .-----.
490088 // | - |FT1|FT0|ARM| LS|LAS|BOS|WTS|
490089 // |-----|
490090 // | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0x48
490091 // `-----'
490092 // \_____/ : | : : :
490093 // | : | : : Byte DMA transfer
490094 // | : | : :
490095 // | : | : Little endian byte order
490096 // | : | :

```

```
490097 // | : | Dual 16 bit DMA mode
490098 // | : |
490099 // | : Loopback OFF (normal operation)
490100 // | :
490101 // | Send Command non executed: all frames removed
490102 // from
490103 // | Buffer Ring under program control
490104 // |
490105 // FIFO threshold 8 bytes
490106 //
490107 out_8 ((io + NE2K_DCR), 0x48);
490108 //
490109 // Reset remote byte count registers.
490110 //
490111 out_8 ((io + NE2K_RBCR0), 0x00);
490112 out_8 ((io + NE2K_RBCR1), 0x00);
490113 //
490114 // Disable interrupts with an empty mask.
490115 //
490116 out_8 ((io + NE2K_IMR), 0x00);
490117 //
490118 // Reset all ISR register flags.
490119 //
490120 out_8 ((io + NE2K_ISR), 0xFF);
490121 //
490122 // Receive configuration register (RCR)
490123 // .-----'.
490124 // | - | - |MON|PRO| AM| AB| AR|SEP|
490125 // |-----|
490126 // | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0x20
490127 // `-----'
490128 // | : : : : :
490129 // | : : : : Frames with receive errors are
490130 // | : : : : rejected
490131 // | : : : :
490132 // | : : : Frames with fewer than 64 bytes rejected
490133 // | : : :
```

```

490134 // | : : Frames with broadcast destination rejected
490135 // | : : accepted
490136 // | : :
490137 // | : Frames with multicast destination address
490138 // | : not checked
490139 // | :
490140 // | Physical address of node must match the station
490141 // | address
490142 // |
490143 // Monitor mode: frames checked but not buffered to
490144 // memory
490145 //
490146 // Monitor mode.
490147 //
490148 out_8 ((io + NE2K_RCR), 0x20);
490149 //
490150 // Transmit configuration register (TCR)
490151 // .------.
490152 // | - | - | - | OFST|ATD|LB1|LB0|CRC|
490153 // |-----|
490154 // | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0x02
490155 // `-----'
490156 // \_____/ :
490157 // | CRC appended by the transmitter
490158 // |
490159 // Internal loopback (mode 1)
490160 //
490161 // [Loopback is not supported by Bochs]
490162 //
490163 // Transmit loopback.
490164 //
490165 out_8 ((io + NE2K_TCR), 0x02);
490166 //
490167 // Remote byte count registers to NE2K_SAPROM_SIZE:
490168 // the bytes to be
490169 // read from SA-PROM.
490170 //

```



```
490171 out_8 ((io + NE2K_RBCR0), NE2K_SAPROM_SIZE);
490172 out_8 ((io + NE2K_RBCR1), (NE2K_SAPROM_SIZE >> 8));
490173 //
490174 // Set the remote DMA address to zero.
490175 //
490176 out_8 ((io + NE2K_RSAR0), 0x00); // Must be
490177 // zero.
490178 out_8 ((io + NE2K_RSAR1), 0x00);
490179 //
490180 // Command register (CR)
490181 // .------.
490182 // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
490183 // |-----|
490184 // | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
490185 // `-----'
490186 // \___/ \___/ |
490187 // | | START
490188 // | Read
490189 // |
490190 // Register
490191 // page 0
490192 //
490193 out_8 ((io + NE2K_CR), 0x0A);
490194 //
490195 // Save the SA-PROM content.
490196 //
490197 for (i = 0; i < NE2K_SAPROM_SIZE; i++)
490198 {
490199     sa_prom[i] = in_8 (io + NE2K_DATA);
490200 }
490201 //
490202 // Set NIC physical address from SA-PROM data.
490203 //
490204 par[0] = sa_prom[0];
490205 par[1] = sa_prom[2];
490206 par[2] = sa_prom[4];
490207 par[3] = sa_prom[6];
```

```

490208 par[4] = sa_prom[8];
490209 par[5] = sa_prom[10];
490210 //
490211 // Copy to the 'address' pointer.
490212 //
490213 if (address != NULL)
490214 {
490215     memcpy (address, par, (size_t) 6);
490216 }
490217 //
490218 // -----
490219 // INITIALIZATION SEQUENCE
490220 // -----
490221 //
490222 // Command register (CR)
490223 // .------.
490224 // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
490225 // |-----|
490226 // | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
490227 // '-----'
490228 // \_____/ \_____/ |
490229 // | | STOP
490230 // | |
490231 // | Abort/complete
490232 // | remote DMA
490233 // |
490234 // Register
490235 // page 0
490236 //
490237 out_8 ((io + NE2K_CR), 0x21);
490238 //
490239 // There is no need to check the ISR value. At this
490240 // point,
490241 // ISR might report a reset status or a remote DMA
490242 // complete.
490243 // Go to the DCR register.
490244 //

```

```

490245 // Data configuration register (DCR)
490246 // .------.
490247 // | - |FT1|FT0|ARM| LS|LAS|BOS|WTS|
490248 // |-----|
490249 // | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0x48
490250 // `-----'
490251 // \_____/ : | : : :
490252 // | : | : : Byte DMA transfer
490253 // | : | : :
490254 // | : | : Little endian byte order
490255 // | : | :
490256 // | : | Dual 16 bit DMA mode
490257 // | : |
490258 // | : Loopback OFF (normal operation)
490259 // | :
490260 // | Send Command non executed: all frames removed
490261 // from
490262 // | Buffer Ring under program control
490263 // |
490264 // FIFO threshold 8 bytes
490265 //
490266 out_8 ((io + NE2K_DCR), 0x48);
490267 //
490268 // Reset remote byte count registers.
490269 //
490270 out_8 ((io + NE2K_RBCR0), 0x00);
490271 out_8 ((io + NE2K_RBCR1), 0x00);
490272 //
490273 // Receive configuration register (RCR)
490274 // .------.
490275 // | - | - |MON|PRO| AM| AB| AR|SEP|
490276 // |-----|
490277 // | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0x04
490278 // `-----'
490279 // : : : | : :
490280 // : : : | : Frames with receive errors are
490281 // : : : | : rejected

```

```

490282 // : : : | :
490283 // : : : | Frames with fewer than 64 bytes rejected
490284 // : : : |
490285 // : : : Frames with broadcast destination address
490286 // : : : accepted
490287 // : : :
490288 // : : Frames with multicast destination address
490289 // : : not checked
490290 // : :
490291 // : Physical address of node must match the station
490292 // : address
490293 // :
490294 // Frames buffered to memory
490295 //
490296 // Normal operation and broadcast accepted.
490297 //
490298 out_8 ((io + NE2K_RCR), 0x04);
490299 //
490300 // Transmit page start (local DMA).
490301 //
490302 out_8 ((io + NE2K_TPSR), NE2K_TX_START);
490303 //
490304 // Transmit configuration register (TCR)
490305 // .------.
490306 // | - | - | - |OFST|ATD|LB1|LB0|CRC|
490307 // |-----|
490308 // | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0x02
490309 // '-----'
490310 // \_____/ :
490311 // | CRC appended by the transmitter
490312 // |
490313 // Internal loopback (mode 1)
490314 //
490315 // [Loopback is not supported by Bochs]
490316 //
490317 // Transmit loopback.
490318 //

```

```
490319 out_8 ((io + NE2K_TCR), 0x02);
490320 //
490321 // Set receive buffer page start (local DMA).
490322 //
490323 out_8 ((io + NE2K_PSTART), NE2K_RX_START);
490324 //
490325 // Set boundary: the frame not yet read. At the
490326 // moment, it is the same
490327 // as the receive buffer page start.
490328 //
490329 out_8 ((io + NE2K_BNRY), NE2K_RX_START);
490330 //
490331 // Set receive buffer page stop (local DMA).
490332 //
490333 out_8 ((io + NE2K_PSTOP), NE2K_RX_STOP);
490334 //
490335 // Command register (CR)
490336 // -----
490337 // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
490338 // |-----|
490339 // | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0x61
490340 // `-----'
490341 // \_____/ \_____/ |
490342 // | | STOP
490343 // | |
490344 // | Abort/complete
490345 // | remote DMA
490346 // |
490347 // Register
490348 // page 1
490349 //
490350 out_8 ((io + NE2K_CR), 0x61);
490351 //
490352 // Save physical address and multicast address.
490353 //
490354 out_8 ((io + NE2K_PAR0), par[0]);
490355 out_8 ((io + NE2K_PAR1), par[1]);
```

```
490356 out_8 ((io + NE2K_PAR2), par[2]);
490357 out_8 ((io + NE2K_PAR3), par[3]);
490358 out_8 ((io + NE2K_PAR4), par[4]);
490359 out_8 ((io + NE2K_PAR5), par[5]);
490360 //
490361 out_8 ((io + NE2K_MAR0), 0);
490362 out_8 ((io + NE2K_MAR1), 0);
490363 out_8 ((io + NE2K_MAR2), 0);
490364 out_8 ((io + NE2K_MAR3), 0);
490365 out_8 ((io + NE2K_MAR4), 0);
490366 out_8 ((io + NE2K_MAR5), 0);
490367 out_8 ((io + NE2K_MAR6), 0);
490368 out_8 ((io + NE2K_MAR7), 0);
490369 //
490370 // Set current page: the first frame to be saved
490371 // inside the receive
490372 // buffer. At the moment, it is the same as the
490373 // buffer page start.
490374 //
490375 out_8 ((io + NE2K_CURR), NE2K_RX_START);
490376 //
490377 // Command register (CR)
490378 // .-----.
490379 // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
490380 // |-----|
490381 // | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0x22
490382 // `-----'
490383 // \_____/ \_____/ |
490384 // | | START
490385 // | |
490386 // | Abort/complete
490387 // | remote DMA
490388 // |
490389 // Register
490390 // page 0
490391 //
490392 out_8 ((io + NE2K_CR), 0x22);
```

```
490393 //
490394 // Reset all ISR register flags.
490395 //
490396 out_8 ((io + NE2K_ISR), 0xFF);
490397 //
490398 // ISR will be polled, but received packets will
490399 // fire the IRQ,
490400 // although it is not necessary. So the IMR
490401 // (Interrupt mask register)
490402 // is now set properly with the value 0x01.
490403 //
490404 // Interrupt mask register (IMR)
490405 // -----.
490406 // | -- |RDCE|CNTE|OVWE|TXEE|RXEE|PTXE|PRXE|
490407 // |-----|
490408 // | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0x01
490409 // '-----'
490410 // |
490411 // Enable interrupt when packet
490412 // received
490413 //
490414 out_8 ((io + NE2K_IMR), 0x01);
490415 //
490416 // Transmit configuration register (TCR)
490417 // -----.
490418 // | - | - | - |OFST|ATD|LB1|LB0|CRC|
490419 // |-----|
490420 // | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x00
490421 // '-----'
490422 //
490423 // Normal operation.
490424 //
490425 out_8 ((io + NE2K_TCR), 0x00);
490426 //
490427 return (0);
490428 }
```

94.4.24 kernel/driver/nic/ne2k/ne2k_rx.c



Si veda la sezione [93.16](#).

```
500001 #include <kernel/driver/pci.h>
500002 #include <kernel/net.h>
500003 #include <kernel/driver/nic/ne2k.h>
500004 #include <kernel/ibm_i386.h>
500005 #include <errno.h>
500006 #include <kernel/lib_k.h>
500007 #include <kernel/lib_s.h>
500008 //-----
500009 #define DEBUG 0
500010 //-----
500011 int
500012 ne2k_rx (uintptr_t io)
500013 {
500014     int i;
500015     int bytes;
500016     int curr;
500017     int bnry;
500018     int next;
500019     int frame_status;
500020     int frame_size;
500021     int status;
500022     int n = net_index_eth (0, NULL, io);
500023     net_buffer_eth_t *buffer;
500024     //
500025     // Verify to have found a valid Ethernet device.
500026     //
500027     if (n < 0)
500028     {
500029         errset (ENODEV);
500030         return (-1);
500031     }
500032     //
500033     // Command register (CR)
500034     // .-----.
```



```

500035 // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
500036 // |-----|
500037 // | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0x62
500038 // `-----'
500039 // \____/ \____/ |
500040 // | | START
500041 // | |
500042 // | Abort/complete remote DMA
500043 // |
500044 // Register page 1
500045 //
500046 out_8 ((io + NE2K_CR), 0x62);
500047 //
500048 // Get the current position.
500049 //
500050 curr = in_8 (io + NE2K_CURR);
500051 //
500052 // Command register (CR)
500053 // .-----
500054 // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
500055 // |-----|
500056 // | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0x22
500057 // `-----'
500058 // \____/ \____/ |
500059 // | | START
500060 // | |
500061 // | Abort/complete remote DMA
500062 // |
500063 // Register page 0
500064 //
500065 out_8 ((io + NE2K_CR), 0x22);
500066 //
500067 // Get the boundary.
500068 //
500069 bnry = in_8 (io + NE2K_BNRY);
500070 //
500071 // -----

```

```
500072 // The function is run because at least a frame was
500073 // received:
500074 // if index 'bnry' and index 'curr' are the same,
500075 // all the receive
500076 // ring buffer is to be copied.
500077 // -----
500078 //
500079 // Get all the frames ready from the internal
500080 // buffer.
500081 //
500082 while (1)
500083 {
500084 //
500085 // Find a place inside the frame table.
500086 //
500087 buffer = net_buffer_eth (n);
500088 //
500089 // Check to have a valid buffer pointer.
500090 //
500091 if (buffer == NULL)
500092 {
500093     errset (errno);
500094     return (-1);
500095 }
500096 //
500097 // First read 4 bytes starting from 'bnry'.
500098 //
500099 out_8 ((io + NE2K_RBCR0), 4);
500100 out_8 ((io + NE2K_RBCR1), 0);
500101 //
500102 // Set the remote DMA address to bnry.
500103 //
500104 out_8 ((io + NE2K_RSAR0), 0x00); // Must be
500105 // zero.
500106 out_8 ((io + NE2K_RSAR1), bnry);
500107 //
500108 // Command register (CR)
```

```
500109 // .-----.
500110 // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
500111 // |-----|
500112 // | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
500113 // `-----'
500114 // \_____/ \_____/ |
500115 // | | START
500116 // | |
500117 // | Read
500118 // |
500119 // Register page 0
500120 //
500121 out_8 (io + NE2K_CR, 0x0A);
500122 //
500123 // Frame status
500124 //
500125 frame_status = in_8 (io + NE2K_DATA);
500126 //
500127 // Next frame.
500128 //
500129 next = in_8 (io + NE2K_DATA);
500130 //
500131 // Frame size low.
500132 //
500133 frame_size = in_8 (io + NE2K_DATA);
500134 //
500135 // Frame size high
500136 //
500137 frame_size += (in_8 (io + NE2K_DATA) * 256);
500138 //
500139 // Interrupt status register (ISR)
500140 // .-----.
500141 // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
500142 // |-----|
500143 // | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
500144 // `-----'
500145 // |
```

```
500146 // Remote DMA complete
500147 //
500148 // Verify to have finished with DMA transfer.
500149 //
500150 status = ne2k_isr_expect (io, 0x40);
500151 if (status)
500152     {
500153         errset (errno);
500154         return (-1);
500155     }
500156 //
500157 // Now read again all the frame plus header (the
500158 // initial 4 bytes).
500159 //
500160 buffer->clock = k_clock ();
500161 buffer->size = frame_size - 4;
500162 //
500163 if (DEBUG)
500164     {
500165         k_printf
500166             ("0x%02x[BNRY=0x%02x "
500167              "CURR=0x%02x]0x%02x size=%i\n",
500168              NE2K_RX_START, bnry, curr, NE2K_RX_STOP,
500169              frame_size);
500170     }
500171 //
500172 if (next == bnry)
500173     {
500174         k_printf
500175             ("[%s] next==bnry but should "
500176              "not happen!\n", __func__);
500177         errset (E_DRIVER_FAULT);
500178         return (-1);
500179     }
500180 //
500181 if (next > bnry)
500182     {
```

```

500183         bytes = frame_size;
500184     }
500185     //
500186     if (next < bnry)
500187     {
500188         //
500189         // Read up to the bottom.
500190         //
500191         bytes = ((NE2K_RX_STOP - bnry) * 256);
500192         bytes = min (bytes, frame_size);
500193     }
500194     //
500195     // Read frame content: first part.
500196     //
500197     out_8 ((io + NE2K_RBCR0), bytes & 0xFF);
500198     out_8 ((io + NE2K_RBCR1), bytes >> 8);
500199     //
500200     out_8 ((io + NE2K_RSAR0), 0);      // MUST be
500201     // zero. :-(
500202     out_8 ((io + NE2K_RSAR1), bnry);
500203     //
500204     // Command register (CR)
500205     // .-----'.
500206     // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
500207     // |-----|
500208     // | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
500209     // '-----'
500210     // \_____/ \_____/ |
500211     // | | START
500212     // | |
500213     // | Read
500214     // |
500215     // Register page 0
500216     //
500217     out_8 (io + NE2K_CR, 0x0A);
500218     //
500219     // Jump the first four bytes (no way to start

```

```
500220 // after
500221 // the page start).
500222 //
500223 in_8 (io + NE2K_DATA);
500224 in_8 (io + NE2K_DATA);
500225 in_8 (io + NE2K_DATA);
500226 in_8 (io + NE2K_DATA);
500227 bytes -= 4;
500228 //
500229 // Get the frame data.
500230 //
500231 i = 0;
500232 for (; bytes > 0; i++, bytes--)
500233 {
500234     buffer->frame.octet[i] = in_8 (io + NE2K_DATA);
500235 }
500236 //
500237 // Interrupt status register (ISR)
500238 // .-----
500239 // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
500240 // |-----|
500241 // | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
500242 // '-----'
500243 // |
500244 // Remote DMA complete
500245 //
500246 // Verify to have finished with DMA transfer.
500247 //
500248 status = ne2k_isr_expect (io, 0x40);
500249 if (status)
500250 {
500251     errset (errno);
500252     return (-1);
500253 }
500254 //
500255 if (next < bnry)
500256 {
```

```

500257 //
500258 // There might be a second part to read.
500259 //
500260 bytes =
500261     frame_size - ((NE2K_RX_STOP - bnry) * 256);
500262 }
500263 //
500264 if (bytes > 0)
500265 {
500266     //
500267     out_8 ((io + NE2K_RBCR0), bytes & 0xFF);
500268     out_8 ((io + NE2K_RBCR1), bytes >> 8);
500269     //
500270     out_8 ((io + NE2K_RSAR0), 0);
500271     out_8 ((io + NE2K_RSAR1), NE2K_RX_START);
500272     //
500273     // Command register (CR)
500274     // -----
500275     // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
500276     // |-----|
500277     // | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
500278     // '-----'
500279     // \____/ \_____/ |
500280     // | | START
500281     // | |
500282     // | Read
500283     // |
500284     // Register page 0
500285     //
500286     out_8 (io + NE2K_CR, 0x0A);
500287     //
500288     for (; bytes > 0; i++, bytes--)
500289     {
500290         buffer->frame.octet[i] =
500291             in_8 (io + NE2K_DATA);
500292     }
500293     //

```

```
500294 // Interrupt status register (ISR)
500295 // .-----.
500296 // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
500297 // |-----|
500298 // | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
500299 // `-----'
500300 // |
500301 // Remote DMA complete
500302 //
500303 // Verify to have finished with DMA
500304 // transfer.
500305 //
500306 status = ne2k_isr_expect (io, 0x40);
500307 if (status)
500308     {
500309         errset (errno);
500310         return (-1);
500311     }
500312 }
500313 //
500314 // Update BNRy.
500315 //
500316 bnry = next;
500317 out_8 (io + NE2K_BNRy, bnry);
500318 //
500319 // If the new bnry is equal to curr, the loop is
500320 // finished.
500321 //
500322 if (bnry == curr)
500323     {
500324         //
500325         // finish.
500326         //
500327         return (0);
500328     }
500329 }
```


500330 }

94.4.25 kernel/driver/nic/ne2k/ne2k_rx_reset.c

Si veda la sezione [93.16](#).

```

510001 #include <kernel/driver/pci.h>
510002 #include <kernel/driver/nic/ne2k.h>
510003 #include <kernel/ibm_i386.h>
510004 #include <errno.h>
510005 #include <kernel/lib_k.h>
510006 #include <kernel/lib_s.h>
510007 //-----
510008 #define DEBUG 1
510009 //-----
510010 int
510011 ne2k_rx_reset (uintptr_t io)
510012 {
510013     int status;
510014     //
510015     // Command register (CR)
510016     // .-----
510017     // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
510018     // |-----|
510019     // | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
510020     // `-----'
510021     // \____/ \____/ |
510022     // | | STOP
510023     // | |
510024     // | Abort/complete remote DMA
510025     // |
510026     // Register page 0
510027     //
510028     out_8 ((io + NE2K_CR), 0x21);
510029     //
510030     // Interrupt status register (ISR)
510031     // .-----

```

```

510032 // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
510033 // |-----|
510034 // | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x80
510035 // `-----'
510036 // |
510037 // Reset
510038 //
510039 status = ne2k_isr_expect (io, 0x80);
510040 if (status)
510041 {
510042     errset (errno);
510043     return (-1);
510044 }
510045 //
510046 //
510047 //
510048 out_8 ((io + NE2K_RBCR0), 0);
510049 out_8 ((io + NE2K_RBCR1), 0);
510050 //
510051 // Command register (CR)
510052 // .----- .
510053 // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
510054 // |-----|
510055 // | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0x22
510056 // `-----'
510057 // \_____/ \_____/ |
510058 // | | START
510059 // | |
510060 // | Abort/complete remote DMA
510061 // |
510062 // Register page 0
510063 //
510064 out_8 (io + NE2K_CR, 0x22);
510065 //
510066 return (0);
510067 }

```

94.4.26 kernel/driver/nic/ne2k/ne2k_tx.c



Si veda la sezione [93.16](#).

```

520001 #include <kernel/driver/pci.h>
520002 #include <kernel/driver/nic/ne2k.h>
520003 #include <kernel/ibm_i386.h>
520004 #include <errno.h>
520005 #include <kernel/lib_k.h>
520006 #include <kernel/lib_s.h>
520007 //-----
520008 int
520009 ne2k_tx (uintptr_t io, void *buffer, size_t size)
520010 {
520011     int i;
520012     int status;
520013     uint8_t *b = buffer;
520014     //
520015     // Read the command register to see if the NIC is
520016     // transmitting.
520017     // The value 0x26 tells that the NIC is
520018     // transmitting.
520019     //
520020     // Command register (CR)
520021     // .-----.
520022     // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
520023     // |-----|
520024     // | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0x26
520025     // `-----'
520026     // \____/ \_____/ | |
520027     // | | | Start
520028     // | | Transmit frame
520029     // | Abort/complete
520030     // | remote DMA
520031     // Register
520032     // page 0
520033     //
520034     status = in_8 (io + NE2K_CR);

```

```

520035     if (status == 0x26)
520036     {
520037         errset (EBUSY);
520038         return (-1);
520039     }
520040     //
520041     // Set up the frame size: the size is split into
520042     // RBCR0 and RBCR1
520043     // registers.
520044     //
520045     out_8 ((io + NE2K_RBCR0), (size & 0xFF));
520046     out_8 ((io + NE2K_RBCR1), (size >> 8));
520047     //
520048     // Set the remote DMA address.
520049     //
520050     out_8 ((io + NE2K_RSAR0), 0x00);           // Must be
520051     // zero.
520052     out_8 ((io + NE2K_RSAR1), NE2K_TX_BUFFER);
520053     //
520054     // Command register (CR)
520055     // .------.
520056     // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
520057     // |-----|
520058     // | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0x12
520059     // `-----'
520060     // \_____/ \_____/ |
520061     // | | Start
520062     // | |
520063     // | Write
520064     // Register
520065     // page 0
520066     //
520067     out_8 ((io + NE2K_CR), 0x12);
520068     //
520069     // Write to the data port all the frame.
520070     //
520071     for (i = 0; i < size; i++)

```

```

520072     {
520073         out_8 ((io + NE2K_DATA), b[i]);
520074     }
520075     //
520076     // Interrupt status register (ISR)
520077     // .------.
520078     // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
520079     // |-----|
520080     // | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
520081     // `-----'
520082     // |
520083     // Remote DMA complete
520084     //
520085     // Verify to have finished with DMA transfer.
520086     //
520087     status = ne2k_isr_expect (io, 0x40);
520088     if (status)
520089     {
520090         errset (errno);
520091         return (-1);
520092     }
520093     //
520094     // Set transmit page start, to the transmit buffer.
520095     //
520096     out_8 (io + NE2K_TPSR, NE2K_TX_BUFFER);
520097     //
520098     // Set transmit byte count (frame size).
520099     //
520100     out_8 ((io + NE2K_TBCR0), (size & 0xFF));
520101     out_8 ((io + NE2K_TBCR1), (size >> 8));
520102     //
520103     // Command register (CR)
520104     // .------.
520105     // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
520106     // |-----|
520107     // | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0x26
520108     // `-----'

```

```

520109 // \_____/ \_____/ | |
520110 // | | | Start
520111 // | | Transmit frame
520112 // | Abort/complete remote DMA
520113 // Register
520114 // page 0
520115 //
520116 // Send frame!
520117 //
520118 out_8 ((io + NE2K_CR), 0x26);
520119 //
520120 // Interrupt status register (ISR)
520121 // .------.
520122 // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
520123 // |-----|
520124 // | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
520125 // `-----'
520126 // | |
520127 // | Frame transmitted with no errors
520128 // Transmit error
520129 //
520130 // Wait the end of transmission: might get a good
520131 // transmission
520132 // report, or an error transmission report.
520133 //
520134 status = ne2k_isr_expect (io, 0x0A);
520135 if (status)
520136     {
520137         errset (errno);
520138         return (-1);
520139     }
520140 //
520141 // Transmit status (TSR)
520142 // .------.
520143 // |OWC|CDH| FU|CRS|ABT|COL| - |PTX|
520144 // |-----|
520145 // | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0x38

```

```

520146 // '-----'
520147 // | | |
520148 // | | Transmit aborted
520149 // | Carrier sense lost
520150 // FIFO underrun
520151 //
520152 // Check if there was an error, during transmission.
520153 //
520154 status = in_8 (io + NE2K_TSR);
520155 if (status & 0x38)
520156     {
520157         errset (EIO);
520158         return (-1);
520159     }
520160 //
520161 // Done.
520162 //
520163 return (0);
520164 //
520165 }

```

94.4.27 kernel/driver/pci.h

Si veda la sezione [93.19](#).

```

530001 #ifndef _KERNEL_DRIVER_PCI_H
530002 #define _KERNEL_DRIVER_PCI_H      1
530003 //-----
530004 #include <stdint.h>
530005 #include <sys/types.h>
530006 //-----
530007 //
530008 #define PCI_MAX_DEVICES           8
530009 #define PCI_MAX_BUSES            256    // Fixed.
530010 #define PCI_MAX_SLOTS            32     // Fixed.
530011 //
530012 #define PCI_CONFIG_ADDRESS        0x0CF8

```

```
530013 #define PCI_CONFIG_DATA          0x0CFC
530014 //
530015 // CONFIG_ADDRESS register structure.
530016 //
530017 typedef union
530018 {
530019     uint32_t selector;
530020     struct
530021     {
530022         uint32_t zero:2,
530023             reg:6,
530024             function:3, slot:5, bus:8, reserved:7, enable:1;
530025     };
530026 } pci_address_t;
530027 //
530028 // CONFIG_DATA register structures.
530029 //
530030 typedef union
530031 {
530032     uint32_t r[16];
530033     struct
530034     {
530035         struct
530036         {
530037             uint32_t vendor_id:16, device_id:16;
530038             //
530039             uint32_t command:16, status:16;
530040             //
530041             uint32_t revision_id:8,
530042                 prog_if:8, subclass:8, class_code:8;
530043             //
530044             uint32_t cache_line_size:8,
530045                 latency_timer:8,
530046                 header_type:7, multi_function:1, bist:8;
530047             //
530048             uint32_t bar0;
530049             uint32_t bar1;
```



```
530050     uint32_t bar2;
530051     uint32_t bar3;
530052     uint32_t bar4;
530053     uint32_t bar5;
530054     uint32_t cardbus_cis_pointer;
530055     uint32_t expansion_rom_base_address;
530056     //
530057     uint32_t subsystem_vendor_id:16, subsystem_id:16;
530058     //
530059     uint32_t capabilities_pointer:8, reserved_1:24;
530060     //
530061     uint32_t reserved_2;
530062     //
530063     uint32_t interrupt_line:8,
530064             interrupt_pin:8, min_grant:8, max_latency:8;
530065 };
530066 };
530067 } pci_header_type_00_t;
530068 //
530069 //-----
530070 //
530071 // PCI table row.
530072 //
530073 typedef struct
530074 {
530075     unsigned char bus;
530076     unsigned char slot;
530077     unsigned short int vendor_id;
530078     unsigned short int device_id;
530079     unsigned char class_code;
530080     unsigned char subclass;
530081     unsigned char prog_if;
530082     uintptr_t base_io;
530083     unsigned char irq;
530084 } pci_t;
530085 //
530086 extern pci_t pci_table[PCI_MAX_DEVICES];
```

```
530087 //
530088 //-----
530089 void pci_init (void);
530090 //-----
530091 #endif
```

94.4.28 kernel/driver/pci/pci_init.c

<<

Si veda la sezione [93.19](#).

```
540001 #include <kernel/driver/pci.h>
540002 #include <kernel/ibm_i386.h>
540003 #include <errno.h>
540004 //-----
540005 extern pci_t pci_table[PCI_MAX_DEVICES];
540006 //-----
540007 void
540008 pci_init (void)
540009 {
540010     pci_header_type_00_t pci;
540011     pci_address_t pci_addr;
540012     //
540013     int t;           // PCI table index.
540014     int b;           // PCI bus index.
540015     int s;           // PCI slot index.
540016     int r;           // PCI header register index.
540017     //
540018     // Reset the PCI table.
540019     //
540020     for (t = 0; t < PCI_MAX_DEVICES; t++)
540021     {
540022         pci_table[t].bus = 0;
540023         pci_table[t].slot = 0;
540024         pci_table[t].vendor_id = 0;
540025         pci_table[t].device_id = 0;
540026         pci_table[t].class_code = 0;
540027         pci_table[t].subclass = 0;
```

```
540028     pci_table[t].prog_if = 0;
540029     pci_table[t].base_io = 0;
540030     pci_table[t].irq = 0;
540031     }
540032     //
540033     // Scan PCI buses and slots.
540034     //
540035     t = 0;
540036     //
540037     for (b = 0; b < PCI_MAX_BUSES && t < PCI_MAX_DEVICES; b++)
540038     {
540039         //
540040         // Will not check multi functions devices (we
540041         // are shure that
540042         // we don't have them).
540043         //
540044         for (s = 0;
540045              s < PCI_MAX_SLOTS && t < PCI_MAX_DEVICES; s++)
540046         {
540047             pci_addr.selector = 0;
540048             pci_addr.enable = 1;
540049             pci_addr.bus = b;
540050             pci_addr.slot = s;
540051             //
540052             pci_addr.reg = 0;
540053             out_32 (PCI_CONFIG_ADDRESS, pci_addr.selector);
540054             pci.r[0] = in_32 (PCI_CONFIG_DATA);
540055             //
540056             if (pci.r[0] == 0xFFFFFFFF)
540057             {
540058                 //
540059                 // There is no such bus:slot
540060                 // combination!
540061                 //
540062                 continue;
540063             }
540064             else
```

```
540065     {
540066         for (r = 1; r < 16; r++)
540067             {
540068                 pci_addr.reg = r;
540069                 out_32 (PCI_CONFIG_ADDRESS,
540070                     pci_addr.selector);
540071                 pci.r[r] = in_32 (PCI_CONFIG_DATA);
540072             }
540073     }
540074     //
540075     // We consider only PCI header type 0x00!
540076     //
540077     if (pci.header_type != 0)
540078     {
540079         continue;
540080     }
540081     //
540082     // We do not consider PCI bridge devices!
540083     //
540084     if (pci.class_code == 0x06)
540085     {
540086         continue;
540087     }
540088     //
540089     // Save the device inside the PCI table.
540090     //
540091     pci_table[t].bus = b;
540092     pci_table[t].slot = s;
540093     pci_table[t].vendor_id = pci.vendor_id;
540094     pci_table[t].device_id = pci.device_id;
540095     pci_table[t].class_code = pci.class_code;
540096     pci_table[t].subclass = pci.subclass;
540097     pci_table[t].prog_if = pci.prog_if;
540098     pci_table[t].base_io = pci.bar0 & 0xFFFFFFF0;
540099     pci_table[t].irq = pci.interrupt_line;
540100     //
540101     k_printf ("%s]: %04x:%04x io=%04x irq=%i\n",
```

```

540102         __func__, pci_table[t].vendor_id,
540103         pci_table[t].device_id,
540104         pci_table[t].base_io, pci_table[t].irq);
540105         //
540106         // Next PCI table row.
540107         //
540108         t++;
540109     }
540110 }
540111 }

```

94.4.29 kernel/driver/pci/pci_public.c

Si veda la sezione [93.19](#).

```

550001 #include <kernel/driver/pci.h>
550002 //-----
550003 pci_t pci_table[PCI_MAX_DEVICES];
550004 //-----

```

94.4.30 kernel/driver/screen.h

Si veda la sezione [93.22](#).

```

560001 #ifndef _KERNEL_DRIVER_SCREEN_H
560002 #define _KERNEL_DRIVER_SCREEN_H 1
560003 //-----
560004 #include <restrict.h>
560005 #include <stdint.h>
560006 //-----
560007 // Virtual consoles data and VGA references.
560008 //
560009 #define SCREEN_MAX      4
560010 #define SCREEN_ROWS    25
560011 #define SCREEN_COLS    80
560012 #define SCREEN_CELLS   (SCREEN_ROWS * SCREEN_COLS)
560013 //

```

```
560014 #define VGA_ATTR          0x07
560015 #define VGA_ADDR          0xB8000
560016 #define VGA_CELL          ((uint16_t *) VGA_ADDR)
560017 //
560018 typedef struct
560019 {
560020     uint16_t cell[SCREEN_CELLS]; // [1]
560021     int position;
560022 } screen_t;
560023 //
560024 // [1] Every character on the screen needs another
560025 //      attribute byte.
560026 //
560027 //-----
560028 extern int screen_active;
560029 extern screen_t screen_table[];
560030 //-----
560031 int screen_clear (screen_t * screen);
560032 screen_t *screen_current (void);
560033 void screen_init (void);
560034 int screen_new_line (screen_t * screen);
560035 int screen_number (screen_t * screen);
560036 screen_t *screen_pointer (int scrn);
560037 int screen_putc (screen_t * screen, int c);
560038 int screen_scroll (screen_t * screen);
560039 int screen_select (screen_t * screen);
560040 void screen_update (screen_t * screen);
560041 //-----
560042 #define screen_cell(c, attrib) \
560043     ((uint16_t) c \
560044      | (((uint16_t) attrib) << 8) & 0xFF00)
560045 //-----
560046 #endif
```

94.4.31 kernel/driver/screen/screen_clear.c



Si veda la sezione [93.22](#).

```
570001 #include <kernel/driver/screen.h>
570002 #include <errno.h>
570003 //-----
570004 int
570005 screen_clear (screen_t * screen)
570006 {
570007     int j;
570008     //
570009     // Check argument.
570010     //
570011     if (screen == NULL)
570012     {
570013         errset (EINVAL);
570014         return (-1);
570015     }
570016     //
570017     // Clear the virtual screen.
570018     //
570019     for (j = 0; j < SCREEN_CELLS; j++)
570020     {
570021         screen->cell[j] = screen_cell (' ', VGA_ATTR);
570022     }
570023     //
570024     // Place the cursor at the top.
570025     //
570026     screen->position = 0;
570027     //
570028     // Update the screen if it is the active one.
570029     //
570030     screen_update (screen);
570031     //
570032     // Ok.
570033     //
570034     return (0);
```

```
570035 }
```

94.4.32 kernel/driver/screen/screen_current.c

<<

Si veda la sezione [93.22](#).

```
580001 #include <kernel/driver/screen.h>
580002 //-----
580003 screen_t *
580004 screen_current (void)
580005 {
580006     if (screen_active >= 0 && screen_active < SCREEN_MAX)
580007     {
580008         return &screen_table[screen_active];
580009     }
580010     else
580011     {
580012         return &screen_table[0];
580013     }
580014 }
```

94.4.33 kernel/driver/screen/screen_init.c

<<

Si veda la sezione [93.22](#).

```
590001 #include <kernel/driver/screen.h>
590002 //-----
590003 void
590004 screen_init (void)
590005 {
590006     int i;
590007     int j;
590008     //
590009     // Reset and clear all virtual consoles.
590010     //
590011     for (i = 0; i < SCREEN_MAX; i++)
590012     {
```



```

590013      //
590014      // Reset position.
590015      //
590016      screen_table[i].position = 0;
590017      //
590018      for (j = 0; j < SCREEN_CELLS; j++)
590019          {
590020              screen_table[i].cell[j] =
590021                  screen_cell (' ', VGA_ATTR);
590022          }
590023      }
590024      //
590025      // Select the first screen.
590026      //
590027      screen_active = 0;
590028  }

```

94.4.34 kernel/driver/screen/screen_new_line.c

Si veda la sezione [93.22](#).

```

600001  #include <kernel/driver/screen.h>
600002  #include <errno.h>
600003  //-----
600004  int
600005  screen_new_line (screen_t * screen)
600006  {
600007      int row;
600008      //
600009      // Check argument.
600010      //
600011      if (screen == NULL)
600012          {
600013              errset (EINVAL);
600014              return (-1);
600015          }
600016      //

```



```
600017 // Find row position on screen.
600018 //
600019 row = (screen->position / SCREEN_COLS);
600020 //
600021 // We want to go one row down.
600022 //
600023 row++;
600024 //
600025 // Scroll the screen if necessary.
600026 //
600027 for (; row >= SCREEN_ROWS; row--)
600028 {
600029     screen_scroll (screen);
600030 }
600031 //
600032 // Reset position at the beginning of the line.
600033 //
600034 screen->position = row * SCREEN_COLS;
600035 //
600036 // Update the video if it is the current one. This
600037 // is necessary to
600038 // update the cursor position, if the original
600039 // column was not zero.
600040 //
600041 screen_update (screen);
600042 //
600043 // Ok.
600044 //
600045 return (0);
600046 }
```

94.4.35 kernel/driver/screen/screen_number.c



Si veda la sezione [93.22](#).

```
610001 #include <kernel/driver/screen.h>
610002 #include <stddef.h>
```

```
610003 #include <errno.h>
610004 #include <kernel/lib_k.h>
610005 //-----
610006 int
610007 screen_number (screen_t * screen)
610008 {
610009     ptrdiff_t distance;
610010     int n;
610011     //
610012     if (screen == NULL)
610013     {
610014         errset (EINVAL);
610015         return (-1);
610016     }
610017     //
610018     distance = (void *) screen - (void *) &screen_table[0];
610019     //
610020     n = (distance % (sizeof (screen_t)));
610021     //
610022     if (n != 0)
610023     {
610024         errset (EINVAL); // Invalid pointer placement.
610025         return (-1);
610026     }
610027     //
610028     n = (distance / (sizeof (screen_t)));
610029     //
610030     if (n < 0 || n > SCREEN_MAX)
610031     {
610032         errset (EINVAL); // Pointer outside the screen
610033         // table.
610034         return (-1);
610035     }
610036     //
610037     // If we are here, variable 'n' holds the right
610038     // screen number.
610039     //
```

```
610040     return (n);
610041 }
```

94.4.36 kernel/driver/screen/screen_pointer.c

<<

Si veda la sezione [93.22](#).

```
620001 #include <kernel/driver/screen.h>
620002 #include <errno.h>
620003 //-----
620004 screen_t *
620005 screen_pointer (int scrn)
620006 {
620007     if (scrn >= 0 && scrn < SCREEN_MAX)
620008     {
620009         return &screen_table[scrn];
620010     }
620011     else
620012     {
620013         errset (EINVAL);
620014         return (NULL);
620015     }
620016 }
```

94.4.37 kernel/driver/screen/screen_public.c

<<

Si veda la sezione [93.22](#).

```
630001 #include <kernel/driver/screen.h>
630002 #include <errno.h>
630003 //-----
630004 int screen_active;
630005 screen_t screen_table[SCREEN_MAX];
```

94.4.38 kernel/driver/screen/screen_putc.c



Si veda la sezione [93.22](#).

```
640001 #include <kernel/driver/screen.h>
640002 #include <errno.h>
640003 //-----
640004 int
640005 screen_putc (screen_t * screen, int c)
640006 {
640007     int row;
640008     int col;
640009     //
640010     if (screen == NULL)
640011     {
640012         errset (EINVAL);
640013         return (-1);
640014     }
640015     //
640016     // Find row-col position on screen.
640017     //
640018     row = (screen->position / SCREEN_COLS);
640019     col = (screen->position - (row * SCREEN_COLS));
640020     //
640021     //
640022     //
640023     if (c == '\n' || c == '\r')
640024     {
640025         screen_new_line (screen);
640026         return (0);
640027     }
640028     else if (c == '\b')
640029     {
640030         screen->position--;
640031         if (screen->position < 0)
640032         {
640033             screen->position = 0;
640034         }

```

```

640035     screen_update (screen);
640036     return (0);
640037 }
640038 else if (screen->position == (SCREEN_CELLS - 1))
640039 {
640040     //
640041     // It is not a control character and we are
640042     // already at the
640043     // last cell of the last row.
640044     //
640045     screen_scroll (screen);
640046 }
640047 //
640048 // If we are here, it is not a control character.
640049 // So: print it.
640050 //
640051 screen->cell[screen->position] =
640052     screen_cell (c, VGA_ATTR);
640053 screen->position++;
640054 screen_update (screen);
640055 //
640056 return (0);
640057 }

```

94.4.39 kernel/driver/screen/screen_scroll.c

«

Si veda la sezione [93.22](#).

```

650001 #include <kernel/driver/screen.h>
650002 #include <errno.h>
650003 //-----
650004 int
650005 screen_scroll (screen_t * screen)
650006 {
650007     int a;           // screen[].cell[] index.
650008     int b;           // screen[].cell[] index
650009     //

```

```
650010 // Check argument.
650011 //
650012 if (screen == NULL)
650013 {
650014     errset (EINVAL);
650015     return (-1);
650016 }
650017 //
650018 // Move up a line.
650019 //
650020 for (a = 0, b = SCREEN_COLS; b < SCREEN_CELLS; a++, b++)
650021 {
650022     screen->cell[a] = screen->cell[b];
650023 }
650024 //
650025 // Clear last screen line.
650026 //
650027 for (b = (SCREEN_CELLS - SCREEN_COLS);
650028     b < SCREEN_CELLS; b++)
650029 {
650030     screen->cell[b] = screen_cell (' ', VGA_ATTR);
650031 }
650032 //
650033 // Update position.
650034 //
650035 screen->position -= SCREEN_COLS;
650036 if (screen->position < 0)
650037 {
650038     screen->position = 0;
650039 }
650040 //
650041 // Update the video if it is the current one.
650042 //
650043 screen_update (screen);
650044 //
650045 // Ok.
650046 //
```

```
650047     return (0);
650048 }
```

94.4.40 kernel/driver/screen/screen_select.c

<<

Si veda la sezione [93.22](#).

```
660001 #include <kernel/driver/screen.h>
660002 #include <errno.h>
660003 #include <kernel/ibm_i386.h>
660004 //-----
660005 int
660006 screen_select (screen_t * screen)
660007 {
660008     int scrn;
660009     //
660010     if (screen == NULL)
660011     {
660012         errset (EINVAL);
660013         return (-1);
660014     }
660015     //
660016     // Get screen number.
660017     //
660018     scrn = screen_number (screen);
660019     if (scrn < 0)
660020     {
660021         errset (EINVAL); // The screen pointer was
660022         // invalid.
660023         return (-1);
660024     }
660025     //
660026     // Set the current screen, update the screen memory
660027     // and put the cursor.
660028     //
660029     screen_active = scrn;
660030     //
```



```
660031     screen_update (screen);
660032     //
660033     // Ok.
660034     //
660035     return (0);
660036 }
```

94.4.41 kernel/driver/screen/screen_update.c



Si veda la sezione [93.22](#).

```
670001 #include <kernel/driver/screen.h>
670002 #include <kernel/ibm_i386.h>
670003 #include <stddef.h>
670004 //-----
670005 void
670006 screen_update (screen_t * screen)
670007 {
670008     screen_t *screen_showing;
670009     int j;
670010     unsigned char position_high;
670011     unsigned char position_low;
670012     //
670013     // Check input: if it is the NULL pointer, or it is
670014     // not a valid
670015     // pointer, then select the current screen.
670016     //
670017     if ((screen == NULL) || (screen_number (screen) < 0))
670018     {
670019         screen = screen_current ();
670020     }
670021     //
670022     // Get current screen anyway.
670023     //
670024     screen_showing = screen_current ();
670025     //
670026     // Verify again to be in a valid screen.
```

```
670027 //
670028 if (screen_number (screen_showing) < 0)
670029 {
670030     return;
670031 }
670032 //
670033 // If the selected screen is also the current
670034 // screen, then
670035 // must update the content (otherwise there is
670036 // nothing to do).
670037 //
670038 if (screen_showing == screen)
670039 {
670040     //
670041     // Copy virtual screen to real screen memory.
670042     //
670043     for (j = 0; j < SCREEN_CELLS; j++)
670044     {
670045         VGA_CELL[j] = screen->cell[j];
670046     }
670047     //
670048     // Place the cursor.
670049     //
670050     position_high =
670051         (unsigned char) (screen->position >> 8);
670052     position_low = (unsigned char) (screen->position);
670053     //
670054     out_8 (0x3D4, 0x0E);
670055     out_8 (0x3D5, position_high);
670056     out_8 (0x3D4, 0x0F);
670057     out_8 (0x3D5, position_low);
670058 }
670059 }
```

94.4.42 kernel/driver/tty.h



Si veda la sezione [93.24](#).

```
680001 #ifndef _KERNEL_DRIVER_TTY_H
680002 #define _KERNEL_DRIVER_TTY_H 1
680003 //-----
680004 #include <stddef.h>
680005 #include <stdint.h>
680006 #include <stdio.h>
680007 #include <sys/types.h>
680008 #include <kernel/ibm_i386.h>
680009 #include <termios.h>
680010 //-----
680011 #define TTY_CONSOLE 4
680012 #define TTY_SERIAL 0
680013 #define TTY_TOTAL (TTY_CONSOLE + TTY_SERIAL)
680014 //-----
680015 #define TTY_INPUT_LINE_EDITING 0
680016 #define TTY_INPUT_LINE_CLOSED 1
680017 //-----
680018 typedef struct
680019 {
680020     dev_t device;
680021     pid_t pgrp; // Process group.
680022     struct termios attr; // termios attributes.
680023     unsigned char status; // 0 = edit, 1 = end edit.
680024     char line[MAX_CANON]; // Canonical input line.
680025     int lpr; // Input line position read.
680026     int lpw; // Input line position write.
680027 } tty_t;
680028 //-----
680029 extern tty_t tty_table[TTY_TOTAL];
680030 //-----
680031 tty_t *tty_reference (dev_t device);
680032 dev_t tty_console (dev_t device);
680033 int tty_read (dev_t device);
680034 void tty_write (dev_t device, int c);
```

```
680035 void tty_init (void);
680036 //-----
680037 #endif
```

94.4.43 kernel/driver/tty/tty_console.c

«

Si veda la sezione [93.24](#).

```
690001 #include <sys/os32.h>
690002 #include <kernel/driver/tty.h>
690003 #include <kernel/driver/screen.h>
690004 //-----
690005 dev_t
690006 tty_console (dev_t device)
690007 {
690008     static dev_t device_active = DEV_CONSOLE0;    // First
690009     // time.
690010     dev_t device_previous;
690011     screen_t *screen;
690012     //
690013     // Check if it required only the current device.
690014     //
690015     if (device == 0)
690016     {
690017         return (device_active);
690018     }
690019     //
690020     // Fix if the device is not valid.
690021     //
690022     if (device > DEV_CONSOLE3 || device < DEV_CONSOLE0)
690023     {
690024         device = DEV_CONSOLE0;
690025     }
690026     //
690027     // Update.
690028     //
690029     device_previous = device_active;
```

```
690030     device_active = device;
690031     //
690032     // Get screen pointer.
690033     //
690034     screen = screen_pointer ((int) (device_active & 0x00FF));
690035     //
690036     // Switch.
690037     //
690038     screen_select (screen);
690039     //
690040     // Return previous device value.
690041     //
690042     return (device_previous);
690043 }
```

94.4.44 kernel/driver/tty/tty_init.c

Si veda la sezione [93.24](#).

```
700001 #include <sys/os32.h>
700002 #include <kernel/driver/tty.h>
700003 #include <kernel/driver/screen.h>
700004 #include <termios.h>
700005 //-----
700006 void
700007 tty_init (void)
700008 {
700009     int page;      // console page.
700010     //
700011     // Console initialization: console pages correspond
700012     // to the first
700013     // terminal items.
700014     //
700015     for (page = 0; page < TTYS_CONSOLE; page++)
700016     {
700017         tty_table[page].device = DEV_CONSOLE0 + page;
700018         tty_table[page].pgrp = 0;
```

```
700019     tty_table[page].line[0] = 0;
700020     tty_table[page].lpr = 0;
700021     tty_table[page].lpw = 0;
700022     tty_table[page].status = TTY_INPUT_LINE_EDITING;
700023     //
700024     // Termios default configuration.
700025     //
700026     tty_table[page].attr.c_iflag = BRKINT | ICRNL;
700027     tty_table[page].attr.c_oflag = 0;
700028     tty_table[page].attr.c_cflag = 0;
700029     tty_table[page].attr.c_lflag =
700030         ECHO | ECHOE | ECHOK | ECHONL | ICANON | ISIG;
700031     //
700032     // VEOF == ASCII EOT
700033     //
700034     tty_table[page].attr.c_cc[VEOF] = 0x04;
700035     //
700036     // VEOL == undefined
700037     //
700038     tty_table[page].attr.c_cc[VEOL] = 0x00;
700039     //
700040     // VERASE == ASCII BS
700041     //
700042     tty_table[page].attr.c_cc[VERASE] = 0x08;
700043     //
700044     // VINTR == ASCII ETX
700045     //
700046     tty_table[page].attr.c_cc[VINTR] = 0x03;
700047     //
700048     // VKILL == undefined
700049     //
700050     tty_table[page].attr.c_cc[VKILL] = 0x00;
700051     //
700052     // VMIN == 0
700053     //
700054     tty_table[page].attr.c_cc[VMIN] = 0x00;
700055     //
```

```

700056      // VQUIT == ASCII FS
700057      //
700058      tty_table[page].attr.c_cc[VQUIT] = 0x1C;
700059      //
700060      // VSTART == undefined
700061      //
700062      tty_table[page].attr.c_cc[VSTART] = 0x00;
700063      //
700064      // VSUSP == undefined
700065      //
700066      tty_table[page].attr.c_cc[VSUSP] = 0x00;
700067      //
700068      // VTIME == 0
700069      //
700070      tty_table[page].attr.c_cc[VTIME] = 0x00;
700071      }
700072      //
700073      // Set video mode.
700074      //
700075      screen_init ();
700076      //
700077      // Select the first console.
700078      //
700079      tty_console (DEV_CONSOLE0);
700080      //
700081      // Nothing else to configure (only consoles are
700082      // available).
700083      //
700084      return;
700085      }

```

94.4.45 kernel/driver/tty/tty_public.c

Si veda la sezione [93.24](#).

```

710001 #include <kernel/driver/tty.h>
710002 //-----

```

```
710003 tty_t tty_table[TTYS_TOTAL];
```

94.4.46 kernel/driver/tty/tty_read.c

<<

Si veda la sezione [93.24](#).

```
720001 #include <sys/os32.h>
720002 #include <kernel/driver/tty.h>
720003 #include <kernel/lib_k.h>
720004 //-----
720005 int
720006 tty_read (dev_t device)
720007 {
720008     tty_t *tty;
720009     int key;
720010     //
720011     tty = tty_reference (device);
720012     if (tty == NULL)
720013     {
720014         k_printf
720015             ("kernel alert: cannot find terminal device "
720016              "0x%08x!\n", (int) device);
720017         //
720018         return (-1);
720019     }
720020     //
720021     // Read from canonical input line, but only if it is
720022     // time to read.
720023     //
720024     if (tty->status == TTY_INPUT_LINE_CLOSED)
720025     {
720026         if (tty->lpr > tty->lpw)
720027         {
720028             //
720029             // There is nothing to read!
720030             // Reset input line.
720031             //
```



```

720032         tty->lpw = 0;
720033         tty->lpr = 0;
720034         tty->status = TTY_INPUT_LINE_EDITING;
720035         //
720036         return (-1);
720037     }
720038     //
720039     // Read the key.
720040     //
720041     key = tty->line[tty->lpr];
720042     //
720043     // Move up the read cursor.
720044     //
720045     tty->lpr++;
720046 }
720047 else
720048 {
720049     return (-1);
720050 }
720051 //
720052 // Return the key.
720053 //
720054 return (key);
720055
720056 }
```

94.4.47 kernel/driver/tty/tty_reference.c

Si veda la sezione [93.24](#).

```

730001 #include <kernel/driver/tty.h>
730002 //-----
730003 tty_t *
730004 tty_reference (dev_t device)
730005 {
730006     int t;           // Terminal index.
730007     //
```



```
730008 // If device is zero, a reference to the whole table
730009 // is returned.
730010 //
730011 if (device == 0)
730012 {
730013     return (tty_table);
730014 }
730015 //
730016 // Otherwise, a scan is made to find the selected
730017 // device.
730018 //
730019 for (t = 0; t < TTYS_TOTAL; t++)
730020 {
730021     if (tty_table[t].device == device)
730022     {
730023         //
730024         // Device found. Return the pointer.
730025         //
730026         return (&tty_table[t]);
730027     }
730028 }
730029 //
730030 // No device found!
730031 //
730032 return (NULL);
730033 }
```

94.4.48 kernel/driver/tty/tty_write.c

«

Si veda la sezione [93.24](#).

```
740001 #include <sys/os32.h>
740002 #include <kernel/driver/tty.h>
740003 #include <kernel/driver/screen.h>
740004 //-----
740005 void
740006 tty_write (dev_t device, int c)
```

```

740007 {
740008     screen_t *screen;
740009     //
740010     if ((device & 0xFF00) == (DEV_CONSOLE_MAJOR << 8))
740011     {
740012         //
740013         // Get screen pointer.
740014         //
740015         screen = screen_pointer ((int) (device & 0x00FF));
740016         //
740017         screen_putc (screen, c);
740018     }
740019 }

```

94.5 os32: «kernel/fs.h»

Si veda la sezione [93.6](#).

```

750001 #ifndef _KERNEL_FS_H
750002 #define _KERNEL_FS_H 1
750003 //-----
750004 #include <stdint.h>
750005 #include <sys/types.h>
750006 #include <sys/stat.h>
750007 #include <stdio.h>
750008 #include <limits.h>
750009 #include <kernel/memory.h>
750010 #include <sys/socket.h>
750011 #include <netinet/in.h>
750012 #include <netinet/tcp.h>
750013 #include <kernel/net/ip.h>
750014 #include <kernel/net/tcp.h>
750015 #include <sys/os32.h>
750016 //-----
750017 #define SB_MAX_INODE_BLOCKS      8           // 8*8192
750018                                 // inodes max.
750019 #define SB_MAX_ZONE_BLOCKS      8           // 8*8192

```

```

750020                                     // data-zones
750021                                     // max.
750022 #define SB_BLOCK_SIZE          1024          // Fixed for
750023                                     // Minix file
750024                                     // system.
750025 #define SB_MAX_ZONE_SIZE        4096          // log2 max is
750026                                     // 1.
750027 //-----
750028 //
750029 // blocks * (1024 * 8 / 16)
750030 // = number of bits, divided 16.
750031 //
750032 #define SB_MAP_INODE_SIZE        (SB_MAX_INODE_BLOCKS*512)
750033 #define SB_MAP_ZONE_SIZE        (SB_MAX_ZONE_BLOCKS*512)
750034 //-----
750035 //
750036 // Number of zone pointers contained inside a zone,
750037 // used as an indirect inode list
750038 // (a pointer = 16 bits = 2 bytes).
750039 //
750040 #define INODE_MAX_INDIRECT_ZONES (SB_MAX_ZONE_SIZE/2)
750041 //-----
750042 #define INODE_MAX_REFERENCES      0xFF
750043 //-----
750044 typedef uint16_t zno_t; // Zone number.
750045 //-----
750046 // The structured type 'inode_t' must be pre-declared
750047 // here, because the type sb_t, described before the
750048 // inode structure, has a member pointing to a type
750049 // 'inode_t'. So, must be declared previously the type
750050 // 'inode_t' as made of a type 'struct inode', then the
750051 // structure 'inode' can be described. But for a matter
750052 // of coherence, all other structured data declared
750053 // inside this file follow the same procedure.
750054 //
750055 typedef struct sb sb_t;
750056 typedef struct inode inode_t;

```

```
750057 typedef struct sock sock_t;
750058 typedef struct file file_t;
750059 typedef struct fd fd_t;
750060 typedef struct directory directory_t;
750061 //-----
750062 #define SB_MAX_SLOTS 16 // Handle max 16 file
750063 // systems.
750064
750065 struct sb
750066 { // File system super block:
750067     uint16_t inodes; // inodes available;
750068     uint16_t zones; // zones available (disk
750069 // size);
750070     uint16_t map_inode_blocks; // inode bit map
750071 // blocks;
750072     uint16_t map_zone_blocks; // data-zone bit map
750073 // blocks;
750074     uint16_t first_data_zone; // first data-zone;
750075     uint16_t log2_size_zone; // log_2
750076 // (size_zone/block_size);
750077     uint32_t max_file_size; // max file size in
750078 // bytes;
750079     uint16_t magic_number; // file system magic
750080 // number.
750081 // -----
750082 // Extra management data, not saved inside the file
750083 // system
750084 // super block.
750085 // -----
750086     dev_t device; // FS device [3]
750087     inode_t *inode_mounted_on; // [4]
750088     blksize_t blksize; // Calculated zone size.
750089     int options; // [5]
750090     uint16_t map_inode[SB_MAP_INODE_SIZE];
750091     uint16_t map_zone[SB_MAP_ZONE_SIZE];
750092     char changed;
750093 };
```

```
750094
750095 extern sb_t sb_table[SB_MAX_SLOTS];
750096 //
750097 // [3] the member 'device' must be kept at the same
750098 // position, because it is used to calculate the
750099 // super block header size, saved on disk.
750100 //
750101 // [4] If this pointer is not NULL, the super block is
750102 // related to a device mounted on a directory. The
750103 // inode of such directory is recorded here. Please
750104 // note that it is type 'void *', instead of type
750105 // 'inode_t', because type 'inode_t' is declared
750106 // after type 'sb_t'.
750107 // Please note that the type 'sb_t' is declared
750108 // before the type 'inode_t', but this member
750109 // points to a type 'inode_t'.
750110 // This is the reason because it was necessary to
750111 // declare first the type 'inode_t' as made of
750112 // 'struct inode', to be described later. For
750113 // coherence, all derived type made of structured
750114 // data, are first declared as structure, and then,
750115 // later, described.
750116 //
750117 // [5] Mount options can be only 'MOUNT_DEFAULT' or
750118 // 'MOUNT_RO', as defined inside file
750119 // 'lib/sys/os32.h'.
750120 //
750121 //-----
750122 #define INODE_MAX_SLOTS          (32 * OPEN_MAX)
750123 #define INODE_PIPE_BUFFER_SIZE  18      // (7 dir. + 2
750124                                     // ind.) * 2.
750125 //
750126 struct inode
750127 {      // Inode (32 byte total):
750128     uint16_t mode;          // file type and permissions;
750129     uint16_t uid; // user ID (16 bit);
750130     uint32_t size;         // file size in bytes;
```

```
750131     uint32_t time;           // file data modification
750132     // time;
750133     uint8_t gid;          // group ID (8 bit);
750134     uint8_t links;        // links to the inode;
750135     uint16_t direct[7];   // direct zones;
750136     uint16_t indirect1;   // indirect zones;
750137     uint16_t indirect2;   // double indirect zones.
750138     // -----
750139     // Extra management data, not saved inside the disk
750140     // file system.
750141     // -----
750142     sb_t *sb;             // Inode's super block. [7]
750143     ino_t ino;            // Inode number.
750144     sb_t *sb_attached;    // [8]
750145     blkcnt_t blkcnt;      // Rounded size/blksize.
750146     unsigned char references; // Run time active
750147     // references.
750148     char changed:1,       // 1 == to be saved.
750149     pipe_dir:1; // 0 == read, 1 == write.
750150     unsigned char pipe_off_read; // Pipe read offset.
750151     unsigned char pipe_off_write; // Pipe write offset
750152     unsigned char pipe_ref_read; // Pipe read
750153     // references.
750154     unsigned char pipe_ref_write; // Pipe write
750155     // references
750156 };
750157
750158 extern inode_t inode_table[INODE_MAX_SLOTS];
750159 //
750160 // [7] the member 'sb' must be kept at the same
750161 // position, because it is used to calculate the
750162 // inode header size, saved on disk.
750163 //
750164 // [8] If the inode is a mount point for another
750165 // device, the other super block pointer is saved
750166 // inside 'sb_attached'.
750167 //
```

```
750168 //-----
750169 #define SOCK_MAX_SLOTS                64
750170 #define SOCK_MAX_QUEUE                (SOCK_MAX_SLOTS/4)
750171 //
750172 struct sock
750173 {
750174     int family;
750175     int type;
750176     int protocol;
750177     h_addr_t laddr;           // Local address, host byte
750178     // order.
750179     h_port_t lport;          // Local port, host byte
750180     // order.
750181     h_addr_t raddr;          // Remote address, host byte
750182     // order.
750183     h_port_t rport;          // Remote port, host byte
750184     // order.
750185     struct
750186     {
750187         clock_t clock[IP_MAX_PACKETS]; // [9]
750188     } read;
750189     uint8_t active:1,        // Is the socket used?
750190         unreachable:1,      //
750191         unreachable_host:1, // ICMP unreachable status.
750192         unreachable_prot:1, //
750193         unreachable_port:1; //
750194     struct
750195     {
750196         uint16_t conn:4,     // Connection status.
750197             can_write:1,     // Can write to send_data[].
750198             can_read:1,      // Can read from *recv_index.
750199             can_send:1,      // Can send data.
750200             can_recv:1,      // Can receive data.
750201             send_closed:1,   // Closed send direction.
750202             recv_closed:1;   // Closed receive direction.
750203     } //
750204     uint32_t lsq[16];       // Local sequence array.
```



```
750205     uint32_t lsq_ack;    // Expected acknowledge.
750206     uint32_t rsq[16];  // Remote sequence array.
750207     uint8_t lsqi:4,    // Local sequence array index.
750208         rsqi:4;    // Remote sequence array index.
750209     //
750210     clock_t clock;    // When was last send.
750211     //
750212     uint8_t send_data[TCP_MSS - sizeof (struct tcphdr)];
750213     size_t send_size; // Size of 'send_data[]'
750214     // content.
750215     int send_flags;
750216     uint8_t recv_data[TCP_MAX_DATA_SIZE];    // Data
750217     // received.
750218     size_t recv_size; // Size of 'recv_data[]'
750219     // content.
750220     uint8_t *recv_index;    // Read index inside
750221     // 'recv_data[]'.
750222     pid_t listen_pid;    // Process listening at local
750223     // port.
750224     int listen_max;    // Max connection requests.
750225     int listen_queue[SOCK_MAX_QUEUE];    // [10]
750226 } tcp;
750227 };
750228 //
750229 extern sock_t sock_table[SOCK_MAX_SLOTS];
750230 //
750231 // [9] The array 'read.clock[]' has the same size as
750232 // the array as 'ip_tables[]', so that it can be
750233 // saved, inside the former, the clock time of a
750234 // packet read for the socket purposes.
750235 // This is necessary to know if the packet was
750236 // already managed inside the socket system, or
750237 // it is new.
750238 //
750239 // [10] When a process listen o a local port, member
750240 // 'listen_pid' contains the pid number; member
750241 // 'listen_max' contains the max allowed
```

```
750242 //      connections that will be serviced; the array
750243 //      'listen_queue[]' will contain the file
750244 //      descriptors of established connections.
750245 //      If 'listen_queue[x]' is equal to -1, it means
750246 //      that there is no file descriptor there.
750247 //
750248 //-----
750249 #define FILE_MAX_SLOTS          (64 * OPEN_MAX)
750250
750251 struct file
750252 {
750253     int references;
750254     off_t offset; // File position.
750255     int oflags; // Open mode: r/w/r+w [11]
750256     inode_t *inode;
750257     sock_t *sock;
750258 };
750259
750260 extern file_t file_table[FILE_MAX_SLOTS];
750261 //
750262 // [11] the member 'oflags' can get only O_RDONLY,
750263 //      O_WRONLY, O_RDWR, (from header 'fcntl.h')
750264 //      combined with OR binary operator.
750265 //
750266 //-----
750267 struct fd
750268 {
750269     int fl_flags; // File status flags and file
750270 //      access modes. [12]
750271     int fd_flags; // File descriptor flags:
750272 //      currently only FD_CLOEXEC.
750273     file_t *file; // Pointer to the file table.
750274 };
750275 //
750276 // [12] the member 'fl_flags' can get only O_RDONLY,
750277 //      O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_NOCTTY,
750278 //      O_TRUNC, O_APPEND and O_NONBLOCK
```

```
750279 //      (from header 'fcntl.h') combined with OR
750280 //      binary operator.
750281 //      Options like O_DSYNC, O_RSYNC and O_SYNC are
750282 //      not taken into consideration by os32.
750283 //
750284 //      Please notice that each process has its own 'fd'
750285 //      table, embedded inside the process table.
750286 //-----
750287 struct directory
750288 {      // Directory entry:
750289     uint16_t ino; // inode number;
750290     char name[NAME_MAX]; // file name.
750291 };
750292 //-----
750293 void fs_init (void);
750294 //-----
750295 int sb_inode_status (sb_t * sb, ino_t ino);
750296 sb_t *sb_mount (dev_t device, inode_t ** inode_mnt,
750297                int options);
750298 void sb_print (void);
750299 sb_t *sb_reference (dev_t device);
750300 int sb_save (sb_t * sb);
750301 int sb_zone_status (sb_t * sb, zno_t zone);
750302 //-----
750303 zno_t zone_alloc (sb_t * sb);
750304 int zone_free (sb_t * sb, zno_t zone);
750305 void zone_print (sb_t * sb, zno_t zone);
750306 int zone_read (sb_t * sb, zno_t zone, void *buffer);
750307 int zone_write (sb_t * sb, zno_t zone, void *buffer);
750308 //-----
750309 inode_t *inode_alloc (dev_t device, mode_t mode,
750310                     uid_t uid, gid_t gid);
750311 int inode_check (inode_t * inode, mode_t type,
750312                int perm, uid_t uid, gid_t gid);
750313 int inode_dir_empty (inode_t * inode);
750314 ssize_t inode_file_read (inode_t * inode, off_t offset,
750315                          void *buffer, size_t count,
```

```
750316         int *eof);
750317 ssize_t inode_file_write (inode_t * inode,
750318         off_t offset,
750319         const void *buffer, size_t count);
750320 int inode_free (inode_t * inode);
750321 blkcnt_t inode_fzones_read (inode_t * inode,
750322         zno_t zone_start,
750323         void *buffer, blkcnt_t blkcnt);
750324 blkcnt_t inode_fzones_write (inode_t * inode,
750325         zno_t zone_start,
750326         void *buffer, blkcnt_t blkcnt);
750327 inode_t *inode_get (dev_t device, ino_t ino);
750328 inode_t *inode_pipe_make (void);
750329 ssize_t inode_pipe_read (inode_t * inode, void *buffer,
750330         size_t count, int *eof);
750331 ssize_t inode_pipe_write (inode_t * inode,
750332         const void *buffer, size_t count);
750333 void inode_print (void);
750334 int inode_put (inode_t * inode);
750335 inode_t *inode_reference (dev_t device, ino_t ino);
750336 int inode_save (inode_t * inode);
750337 inode_t *inode_stdio_dev_make (dev_t device, mode_t mode);
750338 int inode_truncate (inode_t * inode);
750339 zno_t inode_zone (inode_t * inode, zno_t fzone, int write);
750340 //-----
750341 file_t *file_pipe_make (void);
750342 file_t *file_reference (int fno);
750343 file_t *file_stdio_dev_make (dev_t device, mode_t mode,
750344         int oflags);
750345 //-----
750346 dev_t path_device (pid_t pid, const char *path);
750347 int path_fix (char *path);
750348 int path_full (const char *path,
750349         const char *path_cwd, char *full_path);
750350 inode_t *path_inode (pid_t pid, const char *path);
750351 inode_t *path_inode_link (pid_t pid, const char *path,
750352         inode_t * inode, mode_t mode);
```

```

750353 //-----
750354 int fd_dup (pid_t pid, int fdn_old, int fdn_min);
750355 fd_t *fd_reference (pid_t pid, int *fdn);
750356 //-----
750357 //
750358 // void sock_put (sock_t *s);
750359 //
750360 #define sock_put(s) (s->active=0)
750361
750362 sock_t *sock_reference (int skn);
750363 h_port_t sock_free_port (void);
750364 //-----
750365
750366 #endif

```

94.5.1	kernel/fs/fd_dup.c	1159
94.5.2	kernel/fs/fd_reference.c	1161
94.5.3	kernel/fs/file_pipe_make.c	1162
94.5.4	kernel/fs/file_reference.c	1163
94.5.5	kernel/fs/file_stdio_dev_make.c	1164
94.5.6	kernel/fs/fs_init.c	1166
94.5.7	kernel/fs/fs_public.c	1167
94.5.8	kernel/fs/inode_alloc.c	1167
94.5.9	kernel/fs/inode_check.c	1172
94.5.10	kernel/fs/inode_dir_empty.c	1175
94.5.11	kernel/fs/inode_file_read.c	1177
94.5.12	kernel/fs/inode_file_write.c	1181
94.5.13	kernel/fs/inode_free.c	1184

94.5.14	kernel/fs/inode_fzones_read.c	1185
94.5.15	kernel/fs/inode_fzones_write.c	1187
94.5.16	kernel/fs/inode_get.c	1189
94.5.17	kernel/fs/inode_pipe_make.c	1195
94.5.18	kernel/fs/inode_pipe_read.c	1197
94.5.19	kernel/fs/inode_pipe_write.c	1200
94.5.20	kernel/fs/inode_print.c	1203
94.5.21	kernel/fs/inode_put.c	1206
94.5.22	kernel/fs/inode_reference.c	1208
94.5.23	kernel/fs/inode_save.c	1211
94.5.24	kernel/fs/inode_stdio_dev_make.c	1213
94.5.25	kernel/fs/inode_truncate.c	1215
94.5.26	kernel/fs/inode_zone.c	1219
94.5.27	kernel/fs/path_device.c	1234
94.5.28	kernel/fs/path_fix.c	1235
94.5.29	kernel/fs/path_full.c	1237
94.5.30	kernel/fs/path_inode.c	1239
94.5.31	kernel/fs/path_inode_link.c	1245
94.5.32	kernel/fs/sb_inode_status.c	1253
94.5.33	kernel/fs/sb_mount.c	1255
94.5.34	kernel/fs/sb_print.c	1260
94.5.35	kernel/fs/sb_reference.c	1261

94.5.36	kernel/fs/sb_save.c	1263
94.5.37	kernel/fs/sb_zone_status.c	1265
94.5.38	kernel/fs/sock_free_port.c	1266
94.5.39	kernel/fs/sock_reference.c	1267
94.5.40	kernel/fs/zone_alloc.c	1268
94.5.41	kernel/fs/zone_free.c	1271
94.5.42	kernel/fs/zone_print.c	1273
94.5.43	kernel/fs/zone_read.c	1274
94.5.44	kernel/fs/zone_write.c	1275

94.5.1 kernel/fs/fd_dup.c

Si veda la sezione [93.6.1](#).



```
760001 #include <kernel/proc.h>
760002 #include <kernel/fs.h>
760003 #include <errno.h>
760004 #include <fcntl.h>
760005 //-----
760006 int
760007 fd_dup (pid_t pid, int fdn_old, int fdn_min)
760008 {
760009     proc_t *ps;
760010     int fdn_new;
760011     //
760012     // Verify argument.
760013     //
760014     if (fdn_min < 0 || fdn_min >= OPEN_MAX)
760015     {
760016         errset (EINVAL); // Invalid argument.
760017         return (-1);
760018     }
```

```
760019 //
760020 // Get process.
760021 //
760022 ps = proc_reference (pid);
760023 //
760024 // Verify if 'fdn_old' is a valid value.
760025 //
760026 if (fdn_old < 0 ||
760027     fdn_old >= OPEN_MAX || ps->fd[fdn_old].file == NULL)
760028 {
760029     errset (EBADF); // Bad file descriptor.
760030     return (-1);
760031 }
760032 //
760033 // Find the first free slot and duplicate the file
760034 // descriptor.
760035 //
760036 for (fdn_new = fdn_min; fdn_new < OPEN_MAX; fdn_new++)
760037 {
760038     if (ps->fd[fdn_new].file == NULL)
760039     {
760040         ps->fd[fdn_new].fl_flags =
760041             ps->fd[fdn_old].fl_flags;
760042         ps->fd[fdn_new].fd_flags =
760043             ps->fd[fdn_old].fd_flags & ~FD_CLOEXEC;
760044         ps->fd[fdn_new].file = ps->fd[fdn_old].file;
760045         ps->fd[fdn_new].file->references++;
760046         return (fdn_new);
760047     }
760048 }
760049 //
760050 // No fd slot available.
760051 //
760052 errset (EMFILE); // Too many open files.
760053 return (-1);
760054 }
```


94.5.2 kernel/fs/fd_reference.c



Si veda la sezione [93.6.2](#).

```
770001 #include <kernel/proc.h>
770002 #include <kernel/lib_k.h>
770003 #include <errno.h>
770004 //-----
770005 fd_t *
770006 fd_reference (pid_t pid, int *fdn)
770007 {
770008     proc_t *ps;
770009     //
770010     // Get process.
770011     //
770012     ps = proc_reference (pid);
770013     //
770014     // See what to do.
770015     //
770016     if (*fdn < 0)
770017     {
770018         //
770019         // Find the first free slot.
770020         //
770021         for (*fdn = 0; *fdn < OPEN_MAX; (*fdn)++)
770022         {
770023             if (ps->fd[*fdn].file == NULL)
770024             {
770025                 return (&(ps->fd[*fdn]));
770026             }
770027         }
770028         *fdn = -1;
770029         return (NULL);
770030     }
770031     else
770032     {
770033         if (*fdn < OPEN_MAX)
770034         {
```

```
770035         //
770036         // Might return even a free file descriptor.
770037         //
770038         return (&(ps->fd[*fdn]));
770039     }
770040     else
770041     {
770042         return (NULL);
770043     }
770044 }
770045 }
```

94.5.3 kernel/fs/file_pipe_make.c

<<

Si veda la sezione [93.6.4](#).

```
780001 #include <kernel/proc.h>
780002 #include <errno.h>
780003 #include <fcntl.h>
780004 //-----
780005 file_t *
780006 file_pipe_make (void)
780007 {
780008     inode_t *inode;
780009     file_t *file;
780010     //
780011     // Try to allocate a device inode.
780012     //
780013     inode = inode_pipe_make ();
780014     if (inode == NULL)
780015     {
780016         //
780017         // Variable 'errno' is already set by
780018         // 'inode_stdio_dev_make()'.
780019         //
780020     }
780021     errset (errno);
780022     return (NULL);
```

```

780022     }
780023     //
780024     // Inode allocated: need to allocate the system file
780025     // item.
780026     //
780027     file = file_reference (-1);
780028     if (file == NULL)
780029     {
780030         //
780031         // Remove the inode and return an error.
780032         //
780033         inode_put (inode);
780034         errset (ENFILE); // Too many files open in
780035         // system.
780036         return (NULL);
780037     }
780038     //
780039     // Fill with data the system file item.
780040     //
780041     file->references = 2;
780042     file->oflags = (O_RDONLY | O_WRONLY);
780043     file->inode = inode;
780044     //
780045     // Return system file pointer.
780046     //
780047     return (file);
780048 }

```

94.5.4 kernel/fs/file_reference.c

Si veda la sezione [93.6.5](#).

```

790001 #include <kernel/proc.h>
790002 #include <errno.h>
790003 #include <fcntl.h>
790004 //-----
790005 file_t *

```

```
790006 file_reference (int fno)
790007 {
790008     //
790009     // Check type of request.
790010     //
790011     if (fno < 0)
790012     {
790013         //
790014         // Find a free slot.
790015         //
790016         for (fno = 0; fno < FILE_MAX_SLOTS; fno++)
790017         {
790018             if (file_table[fno].references <= 0)
790019             {
790020                 return (&file_table[fno]);
790021             }
790022         }
790023         return (NULL);
790024     }
790025     else if (fno > FILE_MAX_SLOTS)
790026     {
790027         return (NULL);
790028     }
790029     else
790030     {
790031         return (&file_table[fno]);
790032     }
790033 }
```

94.5.5 kernel/fs/file_stdio_dev_make.c



Si veda la sezione [93.6.6](#).

```
800001 #include <kernel/proc.h>
800002 #include <errno.h>
800003 #include <fcntl.h>
800004 //-----
```

```
800005 file_t *
800006 file_stdio_dev_make (dev_t device, mode_t mode, int oflags)
800007 {
800008     inode_t *inode;
800009     file_t *file;
800010     //
800011     // Try to allocate a device inode.
800012     //
800013     inode = inode_stdio_dev_make (device, mode);
800014     if (inode == NULL)
800015     {
800016         //
800017         // Variable 'errno' is already set by
800018         // 'inode_stdio_dev_make()'.
800019         //
800020         errset (errno);
800021         return (NULL);
800022     }
800023     //
800024     // Inode allocated: need to allocate the system file
800025     // item.
800026     //
800027     file = file_reference (-1);
800028     if (file == NULL)
800029     {
800030         //
800031         // Remove the inode and return an error.
800032         //
800033         inode_put (inode);
800034         errset (ENFILE); // Too many files open in
800035         // system.
800036         return (NULL);
800037     }
800038     //
800039     // Fill with data the system file item.
800040     //
800041     file->references = 1;
```

```
800042 file->oflags = (oflags & (O_RDONLY | O_WRONLY));
800043 file->inode = inode;
800044 //
800045 // Return system file pointer.
800046 //
800047 return (file);
800048 }
```

94.5.6 kernel/fs/fs_init.c

«

Si veda la sezione [93.6.3](#).

```
810001 #include <kernel/fs.h>
810002 #include <string.h>
810003 //-----
810004 void
810005 fs_init (void)
810006 {
810007     int s;
810008     int i;
810009     int f;
810010     //
810011     for (s = 0; s < SB_MAX_SLOTS; s++)
810012     {
810013         sb_table[s].device = 0;
810014         sb_table[s].inode_mounted_on = NULL;
810015     }
810016     //
810017     for (i = 0; i < INODE_MAX_SLOTS; i++)
810018     {
810019         inode_table[i].references = 0;
810020     }
810021     //
810022     for (f = 0; f < FILE_MAX_SLOTS; f++)
810023     {
810024         file_table[f].references = 0;
810025         file_table[f].inode = NULL;
810026     }
810027 }
```

```

810026         file_table[f].sock = NULL;
810027     }
810028     //
810029     // Reset the socket table with 0x00.
810030     //
810031     memset (sock_table, 0x00, sizeof (sock_table));
810032 }

```

94.5.7 kernel/fs/fs_public.c

Si veda la sezione [93.6](#).

```

820001 #include <kernel/fs.h>
820002 //-----
820003 sb_t sb_table[SB_MAX_SLOTS];
820004 file_t file_table[FILE_MAX_SLOTS];
820005 inode_t inode_table[INODE_MAX_SLOTS];
820006 sock_t sock_table[SOCK_MAX_SLOTS];
820007 //-----

```

94.5.8 kernel/fs/inode_alloc.c

Si veda la sezione [93.6.7](#).

```

830001 #include <kernel/fs.h>
830002 #include <errno.h>
830003 #include <kernel/lib_k.h>
830004 #include <kernel/lib_s.h>
830005 //-----
830006 inode_t *
830007 inode_alloc (dev_t device, mode_t mode, uid_t uid,
830008             gid_t gid)
830009 {
830010     sb_t *sb;
830011     inode_t *inode;
830012     int m;           // Index inside the inode map.
830013     int map_element;

```

```
830014 int map_bit;
830015 int map_mask;
830016 ino_t ino;
830017 //
830018 // Check for arguments.
830019 //
830020 if (mode == 0)
830021 {
830022     errset (EINVAL); // Invalid argument.
830023     return (NULL);
830024 }
830025 //
830026 // Get the super block from the known device.
830027 //
830028 sb = sb_reference (device);
830029 if (sb == NULL)
830030 {
830031     errset (ENODEV); // No such device.
830032     return (NULL);
830033 }
830034 //
830035 // Find a free inode.
830036 //
830037 while (1)
830038 {
830039     //
830040     // Scan the inode bit map, to find a free inode
830041     // for new allocation.
830042     //
830043     for (m = 0; m < (SB_MAP_INODE_SIZE * 16); m++)
830044     {
830045         map_element = m / 16;
830046         map_bit = m % 16;
830047         map_mask = 1 << map_bit;
830048         if (!(sb->map_inode[map_element] & map_mask))
830049             {
830050                 //
```



```
830051         // Found a free element: change the map
830052         // to
830053         // allocate the inode.
830054         //
830055         sb->map_inode[map_element] |= map_mask;
830056         sb->changed = 1;
830057         ino = m; // Found a free inode:
830058         break;  // exit the scan loop.
830059     }
830060 }
830061 //
830062 // Check if the scan was successful.
830063 //
830064 if (ino == 0)
830065 {
830066     errset (ENOSPC); // No space left on
830067     // device.
830068     return (NULL);
830069 }
830070 //
830071 // The inode was allocated inside the map in
830072 // memory.
830073 //
830074 inode = inode_get (device, ino);
830075 if (inode == NULL)
830076 {
830077     errset (ENFILE); // Too many files open
830078     // in system.
830079     return (NULL);
830080 }
830081 //
830082 // Verify if the inode is really free: if it
830083 // isn't, must save
830084 // it to disk.
830085 //
830086 if (inode->size > 0 || inode->links > 0)
830087 {
```

```
830088 //
830089 // Strange: should not have a size! Check if
830090 // there are even
830091 // links. Please note that 255 links (that
830092 // is -1) is to be
830093 // considered a free inode, marked in a
830094 // special way for some
830095 // unknown reason. Currently, 'LINK_MAX' is
830096 // equal to 254,
830097 // for that reason.
830098 //
830099 if (inode->links > 0 && inode->links < LINK_MAX)
830100 {
830101 //
830102 // Tell something.
830103 //
830104 k_printf ("kernel alert: device %04x: "
830105          "found \"free\" inode %i "
830106          "that still has size %i "
830107          "and %i links!\n",
830108          device, ino, inode->size,
830109          inode->links);
830110 //
830111 // The inode must be set again to free,
830112 // inside
830113 // the bit map.
830114 //
830115 map_element = ino / 16;
830116 map_bit = ino % 16;
830117 map_mask = 1 << map_bit;
830118 sb->map_inode[map_element] &= ~map_mask;
830119 sb->changed = 1;
830120 //
830121 // Try to fix: reset all to zero.
830122 //
830123 inode->mode = 0;
830124 inode->uid = 0;
```

```
830125         inode->gid = 0;
830126         inode->time = 0;
830127         inode->links = 0;
830128         inode->size = 0;
830129         inode->direct[0] = 0;
830130         inode->direct[1] = 0;
830131         inode->direct[2] = 0;
830132         inode->direct[3] = 0;
830133         inode->direct[4] = 0;
830134         inode->direct[5] = 0;
830135         inode->direct[6] = 0;
830136         inode->indirect1 = 0;
830137         inode->indirect2 = 0;
830138         inode->changed = 1;
830139         //
830140         // Save fixed inode to disk.
830141         //
830142         inode_put (inode);
830143         continue;
830144     }
830145     else
830146     {
830147         //
830148         // Truncate the inode, save and break.
830149         //
830150         inode_truncate (inode);
830151         inode_save (inode);
830152         break;
830153     }
830154 }
830155 else
830156 {
830157     //
830158     // Considering free the inode found.
830159     //
830160     break;
830161 }
```

```
830162     }
830163     //
830164     // Put data inside the inode.
830165     //
830166     inode->mode = mode;
830167     inode->uid = uid;
830168     inode->gid = gid;
830169     inode->size = 0;
830170     inode->time = s_time ((pid_t) 0, NULL);
830171     inode->links = 0;
830172     inode->changed = 1;
830173     //
830174     // Save the inode.
830175     //
830176     inode_save (inode);
830177     //
830178     // Return the inode pointer.
830179     //
830180     return (inode);
830181 }
```

94.5.9 kernel/fs/inode_check.c

<<

Si veda la sezione [93.6.8](#).

```
840001 #include <kernel/fs.h>
840002 #include <errno.h>
840003 #include <kernel/lib_k.h>
840004 //-----
840005 int
840006 inode_check (inode_t * inode, mode_t type, int perm,
840007             uid_t uid, gid_t gid)
840008 {
840009     //
840010     // Ensure that the variable 'type' has only the
840011     // requested file type.
840012     //
```

```
840013     type = (type & S_IFMT);
840014     //
840015     // Check inode argument.
840016     //
840017     if (inode == NULL)
840018     {
840019         errset (EINVAL); // Invalid argument.
840020         return (-1);
840021     }
840022     //
840023     // The inode is not NULL: verify that the inode is
840024     // of a type
840025     // allowed (the parameter 'type' can hold more than
840026     // one
840027     // possibility).
840028     //
840029     if (!(inode->mode & type))
840030     {
840031         errset (E_FILE_TYPE); // The file type is
840032         // not
840033         return (-1); // the expected one.
840034     }
840035     //
840036     // The file type is correct.
840037     //
840038     if (inode->uid != 0 && uid == 0)
840039     {
840040         return (0); // The root user has all
840041         // permissions.
840042     }
840043     //
840044     // The user is not root or the inode is owned by
840045     // root.
840046     //
840047     if (inode->uid == uid)
840048     {
840049         //
```

```
840050 // The user own the inode and must check user
840051 // permissions.
840052 //
840053 perm = (perm << 6);
840054 if ((inode->mode & perm) ^ perm)
840055 {
840056     errset (EACCES); // Permission denied.
840057     return (-1);
840058 }
840059 else
840060 {
840061     return (0);
840062 }
840063 }
840064 //
840065 // The user does not own the inode: the group
840066 // permissions are
840067 // checked.
840068 //
840069 if (inode->gid == gid)
840070 {
840071     //
840072     // The group own the inode and must check user
840073     // permissions.
840074     //
840075     perm = (perm << 3);
840076     if ((inode->mode & perm) ^ perm)
840077     {
840078         errset (EACCES); // Permission denied.
840079         return (-1);
840080     }
840081     else
840082     {
840083         return (0);
840084     }
840085 }
840086 //
```

```

840087 // The user and the group do not own the inode: the
840088 // other
840089 // permissions are checked.
840090 //
840091 if ((inode->mode & perm) ^ perm)
840092 {
840093     errset (EACCES); // Permission denied.
840094     return (-1);
840095 }
840096 else
840097 {
840098     return (0);
840099 }
840100 }

```

94.5.10 kernel/fs/inode_dir_empty.c

Si veda la sezione [93.6.9](#).

```

850001 #include <kernel/fs.h>
850002 #include <errno.h>
850003 #include <kernel/lib_k.h>
850004 //-----
850005 int
850006 inode_dir_empty (inode_t * inode)
850007 {
850008     off_t start;
850009     char buffer[SB_MAX_ZONE_SIZE];
850010     directory_t *dir;
850011     ssize_t size_read;
850012     int d; // Directory buffer index.
850013     //
850014     // Check argument: must be a directory.
850015     //
850016     if (inode == NULL || !S_ISDIR (inode->mode))
850017     {
850018         errset (EINVAL); // Invalid argument.

```



```
850056 //
850057 // Nothing was found; good!
850058 //
850059 return (1); // true
850060 }
```

94.5.11 kernel/fs/inode_file_read.c



Si veda la sezione [93.6.10](#).

```
860001 #include <kernel/fs.h>
860002 #include <errno.h>
860003 #include <kernel/lib_k.h>
860004 //-----
860005 ssize_t
860006 inode_file_read (inode_t * inode, off_t offset,
860007                 void *buffer, size_t count, int *eof)
860008 {
860009     unsigned char *destination = (unsigned char *) buffer;
860010     unsigned char zone_buffer[SB_MAX_ZONE_SIZE];
860011     blkcnt_t blkcnt_read;
860012     off_t off_fzone; // File zone offset.
860013     off_t off_buffer; // Destination buffer offset.
860014     ssize_t size_read; // Byte transfer counter.
860015     zno_t fzone;
860016     off_t off_end;
860017     //
860018     // The inode pointer must be valid, and
860019     // the start byte must be positive.
860020     //
860021     if (inode == NULL || offset < 0)
860022     {
860023         errset (EINVAL); // Invalid argument.
860024         return ((ssize_t) - 1);
860025     }
860026     //
860027     // Check if the start address is inside the file
```

```
860028 // size. This is not
860029 // an error, but zero bytes are read and '*eof' is
860030 // set. Otherwise,
860031 // '*eof' is reset.
860032 //
860033 if (offset >= inode->size)
860034 {
860035     (eof != NULL) ? *eof = 1 : 0;
860036     return (0);
860037 }
860038 else
860039 {
860040     (eof != NULL) ? *eof = 0 : 0;
860041 }
860042 //
860043 // Adjust, if necessary, the size of read, because
860044 // it cannot be
860045 // larger than the actual file size. The variable
860046 // 'off_end' is
860047 // used to calculate the position *after* the
860048 // requested read.
860049 // Remember that the first file position is byte
860050 // zero; so,
860051 // the byte index inside the file goes from zero to
860052 // inode->size -1.
860053 //
860054 off_end = offset;
860055 off_end += count;
860056 if (off_end > inode->size)
860057 {
860058     count = (inode->size - off_end);
860059 }
860060 //
860061 // Read the first file-zone inside the zone buffer.
860062 //
860063 fzone = offset / inode->sb->blksize;
860064 off_fzone = offset % inode->sb->blksize;
```

```
860065 blkcnt_read =
860066     inode_fzones_read (inode, fzone, zone_buffer,
860067                       (blkcnt_t) 1);
860068 if (blkcnt_read <= 0)
860069     {
860070         //
860071         // Sorry!
860072         //
860073
860074         k_printf
860075             ("inode_fzones_read (inode, fzone %i,... )\n",
860076             fzone);
860077
860078         errset (EUNKNOWN);
860079         return (0);         // Zero bytes read!
860080     }
860081 //
860082 // The first file-zone was read: copy it inside the
860083 // destination
860084 // buffer and continue reading the other zones
860085 // needed. Variables
860086 // 'off_buffer' (destination buffer index) and
860087 // 'size_read' (copy
860088 // byte counter) must be reset here. Variable
860089 // 'off_fzone' is already
860090 // set with the initial offset inside 'zone_buffer'.
860091 //
860092 off_buffer = 0;
860093 size_read = 0;
860094 //
860095 while (count)
860096     {
860097         //
860098         // Copy the zone buffer into the destination.
860099         // Variables
860100         // 'off_fzone', 'off_buffer' and 'size_read'
860101         // must not be
```



```
860139         //
860140         errset (EUNKNOWN);
860141         return (size_read);
860142     }
860143 }
860144 }
860145 //
860146 // The requested size was read completely.
860147 //
860148 return (size_read);
860149 }
```

94.5.12 kernel/fs/inode_file_write.c



Si veda la sezione [93.6.11](#).

```
870001 #include <kernel/fs.h>
870002 #include <errno.h>
870003 #include <kernel/lib_k.h>
870004 //-----
870005 ssize_t
870006 inode_file_write (inode_t * inode, off_t offset,
870007                  const void *buffer, size_t count)
870008 {
870009     unsigned char *buffer_source = (unsigned char *) buffer;
870010     unsigned char buffer_zone[SB_MAX_ZONE_SIZE];
870011     off_t off_fzone;         // File zone offset.
870012     off_t off_source;       // Source buffer offset.
870013     ssize_t size_copied;    // Byte transfer counter.
870014     ssize_t size_written;  // Byte written counter.
870015     zno_t fzone;
870016     zno_t zone;
870017     blkcnt_t blkcnt_read;
870018     int status;
870019     //
870020     // The inode pointer must be valid, and
870021     // the start byte must be positive.
```

```
870022 //
870023 if (inode == NULL || offset < 0)
870024 {
870025     errset (EINVAL); // Invalid argument.
870026     return ((ssize_t) - 1);
870027 }
870028 //
870029 // Read a zone, modify it with the source buffer,
870030 // then write it back
870031 // and continue reading and writing other zones if
870032 // needed.
870033 //
870034 for (size_written = 0, off_source = 0, size_copied =
870035     0; count > 0; size_written += size_copied)
870036 {
870037     //
870038     // Read the next file-zone inside the zone
870039     // buffer: the function
870040     // 'inode_zone()' is used to create
870041     // automatically the zone, if
870042     // it does not exist.
870043     //
870044     fzone = offset / inode->sb->blksize;
870045     off_fzone = offset % inode->sb->blksize;
870046     zone = inode_zone (inode, fzone, 1);
870047     if (zone == 0)
870048     {
870049         //
870050         // Return previously written bytes. The
870051         // variable 'errno' is
870052         // already set by 'inode_zone()'.
870053         //
870054         return (size_written);
870055     }
870056     blkcnt_read =
870057         inode_fzones_read (inode, fzone, buffer_zone,
870058             (blkcnt_t) 1);
```

```
870059     if (blkcnt_read <= 0)
870060     {
870061         //
870062         // Even if the value is zero, there is a
870063         // problem reading the
870064         // zone to be overwritten (because
870065         // 'inode_zone()' should
870066         // have already created such zone). The
870067         // variable 'errno' is
870068         // already set by 'inode_fzones_read()'.
870069         //
870070         return ((ssize_t) - 1);
870071     }
870072     //
870073     // The zone was successfully loaded inside the
870074     // buffer: overwrite
870075     // the zone buffer with the source buffer.
870076     //
870077     for (size_copied = 0;
870078         off_fzone < inode->sb->blksize && count > 0;
870079         off_fzone++, off_source++, size_copied++,
870080         count--, offset++)
870081     {
870082         buffer_zone[off_fzone] =
870083             buffer_source[off_source];
870084     }
870085     //
870086     // Save the zone.
870087     //
870088     status = zone_write (inode->sb, zone, buffer_zone);
870089     if (status != 0)
870090     {
870091         //
870092         // Cannot save the zone: return the size
870093         // already written.
870094         // The variable 'errno' is already set by
870095         // 'zone_write()'.
```

```
870096         //
870097         return (size_written);
870098     }
870099     //
870100     // Zone saved: update the file size if necessary
870101     // (and the inode
870102     // too).
870103     //
870104     if (inode->size <= offset)
870105     {
870106         inode->size = offset;
870107         inode->changed = 1;
870108         inode_save (inode);
870109     }
870110 }
870111 //
870112 // All done successfully: return the value.
870113 //
870114 return (size_written);
870115 }
```

94.5.13 kernel/fs/inode_free.c



Si veda la sezione [93.6.12](#).

```
880001 #include <kernel/fs.h>
880002 #include <errno.h>
880003 #include <kernel/lib_k.h>
880004 //-----
880005 int
880006 inode_free (inode_t * inode)
880007 {
880008     int map_element;
880009     int map_bit;
880010     int map_mask;
880011     //
880012     if (inode == NULL)
```



```

880013     {
880014         errset (EINVAL);  // Invalid argument.
880015         return (-1);
880016     }
880017     //
880018     map_element = inode->ino / 16;
880019     map_bit = inode->ino % 16;
880020     map_mask = 1 << map_bit;
880021     //
880022     if (inode->sb->map_inode[map_element] & map_mask)
880023     {
880024         inode->sb->map_inode[map_element] -= map_mask;
880025         inode->sb->changed = 1;
880026     }
880027     //
880028     inode->mode = 0;
880029     inode->uid = 0;
880030     inode->gid = 0;
880031     inode->size = 0;
880032     inode->time = 0;
880033     inode->links = 0;
880034     inode->changed = 1;
880035     inode->references = 0;
880036     //
880037     return (inode_save (inode));
880038 }

```

94.5.14 kernel/fs/inode_fzones_read.c

Si veda la sezione [93.6.13](#).

```

890001 #include <kernel/fs.h>
890002 #include <errno.h>
890003 #include <kernel/lib_k.h>
890004 //-----
890005 blkcnt_t
890006 inode_fzones_read (inode_t * inode, zno_t zone_start,

```

```
890007         void *buffer, blkcnt_t blkcnt)
890008     {
890009         unsigned char *destination = (unsigned char *) buffer;
890010         int status;    // 'zone_read()' return value.
890011         blkcnt_t blkcnt_read; // Zone counter/index.
890012         zno_t zone;
890013         zno_t fzone;
890014         //
890015         // Read the zones into the destination buffer.
890016         //
890017         for (blkcnt_read = 0, fzone = zone_start;
890018             blkcnt_read < blkcnt; blkcnt_read++, fzone++)
890019             {
890020                 //
890021                 // Calculate the zone number, from the
890022                 // file-zone, reading the
890023                 // inode. If a zone is not really allocated, the
890024                 // result is zero
890025                 // and is valid.
890026                 //
890027                 zone = inode_zone (inode, fzone, 0);
890028                 if (zone == ((zno_t) - 1))
890029                     {
890030                         //
890031                         // This is an error. Return the read zones
890032                         // quantity.
890033                         //
890034                         errset (EUNKNOWN);
890035                         return (blkcnt_read);
890036                     }
890037                 //
890038                 // Update the destination buffer pointer.
890039                 //
890040                 destination += (blkcnt_read * inode->sb->blksize);
890041                 //
890042                 // Read the zone inside the destination buffer,
890043                 // but if the zone
```

```

890044     // is zero, a zeroed zone must be filled.
890045     //
890046     if (zone == 0)
890047     {
890048         memset (destination, 0,
890049             (size_t) inode->sb->blksize);
890050     }
890051     else
890052     {
890053         status = zone_read (inode->sb, zone, destination);
890054         if (status != 0)
890055         {
890056             //
890057             // Could not read the requested zone:
890058             // return the zones
890059             // read correctly.
890060             //
890061             errset (EIO);    // I/O error.
890062             return (blkcnt_read);
890063         }
890064     }
890065 }
890066 //
890067 // All zones read correctly inside the buffer.
890068 //
890069 return (blkcnt_read);
890070 }

```

94.5.15 kernel/fs/inode_fzones_write.c

Si veda la sezione [93.6.13](#).

```

900001 #include <kernel/fs.h>
900002 #include <errno.h>
900003 #include <kernel/lib_k.h>
900004 //-----
900005 blkcnt_t

```

```
900006 inode_fzones_write (inode_t * inode, zno_t zone_start,
900007                     void *buffer, blkcnt_t blkcnt)
900008 {
900009     unsigned char *source = (unsigned char *) buffer;
900010     int status;    // 'zone_read()' return value.
900011     blkcnt_t blkcnt_written;    // Written zones
900012     // counter.
900013     zno_t zone;
900014     zno_t fzone;
900015     //
900016     // Write the zones into the destination buffer.
900017     //
900018     for (blkcnt_written = 0, fzone = zone_start;
900019          blkcnt_written < blkcnt; blkcnt_written++, fzone++)
900020     {
900021         //
900022         // Find real zone from file-zone.
900023         //
900024         zone = inode_zone (inode, fzone, 1);
900025         if (zone == 0 || zone == ((zno_t) - 1))
900026         {
900027             //
900028             // Function 'inode_zone()' should allocate
900029             // automatically
900030             // a missing zone and should return a valid
900031             // zone or
900032             // (zno_t) -1. Anyway, even if a zero zone
900033             // is returned,
900034             // it is an error. Return the
900035             // 'blkcnt_written' value.
900036             //
900037             return (blkcnt_written);
900038         }
900039         //
900040         // Update the source buffer pointer for the next
900041         // zone write.
900042         //
```

```

900043     source += (blkcnt_written * inode->sb->blksize);
900044     //
900045     // Write the zone from the buffer content.
900046     //
900047     status = zone_write (inode->sb, zone, source);
900048     if (status != 0)
900049     {
900050         //
900051         // Cannot write the zone. Return
900052         // 'size_written_zone' value.
900053         //
900054         return (blkcnt_written);
900055     }
900056 }
900057 //
900058 // All zones read correctly inside the buffer.
900059 //
900060 return (blkcnt_written);
900061 }

```

94.5.16 kernel/fs/inode_get.c

Si veda la sezione [93.6.15](#).

```

910001 #include <kernel/fs.h>
910002 #include <errno.h>
910003 #include <kernel/lib_k.h>
910004 #include <kernel/dev.h>
910005 //-----
910006 inode_t *
910007 inode_get (dev_t device, ino_t ino)
910008 {
910009     sb_t *sb;
910010     inode_t *inode;
910011     unsigned long int start;
910012     size_t size;
910013     ssize_t n;

```

```
910014     int status;
910015     //
910016     // Verify if the root file system inode was
910017     // requested.
910018     //
910019     if (device == 0 && ino == 1)
910020     {
910021         //
910022         // Get root file system inode.
910023         //
910024         inode = inode_reference (device, ino);
910025         if (inode == NULL)
910026         {
910027             //
910028             // The file system root directory inode is
910029             // not yet loaded:
910030             // get the first super block.
910031             //
910032             sb = sb_reference ((dev_t) 0);
910033             if (sb == NULL || sb->device == 0)
910034             {
910035                 //
910036                 // This error should never happen.
910037                 //
910038                 errset (EUNKNOWN);           // Unknown
910039                 // error.
910040                 return (NULL);
910041             }
910042             //
910043             // Load the file system root directory inode
910044             // (recursive
910045             // call).
910046             //
910047             inode = inode_get (sb->device, (ino_t) 1);
910048             if (inode == NULL)
910049             {
910050                 //
```

```
910051         // This error should never happen.
910052         //
910053         errset (EUNKNOWN);           // Unknown
910054         // error.
910055         return (NULL);
910056     }
910057     //
910058     // Return the directory inode.
910059     //
910060     return (inode);
910061 }
910062 else
910063 {
910064     //
910065     // The file system root directory inode is
910066     // already
910067     // available.
910068     //
910069     if (inode->references >= INODE_MAX_REFERENCES)
910070     {
910071         errset (ENFILE); // Too many files open
910072         // in system.
910073         return (NULL);
910074     }
910075     else
910076     {
910077         inode->references++;
910078         return (inode);
910079     }
910080 }
910081 }
910082 //
910083 // A common device-inode pair was requested: try to
910084 // find an already
910085 // cached inode.
910086 //
910087 inode = inode_reference (device, ino);
```

```
910088     if (inode != NULL)
910089     {
910090         if (inode->references >= INODE_MAX_REFERENCES)
910091         {
910092             errset (ENFILE);      // Too many files open
910093             // in system.
910094             return (NULL);
910095         }
910096     else
910097     {
910098         inode->references++;
910099         return (inode);
910100     }
910101 }
910102 //
910103 // The inode is not yet available: get super block.
910104 //
910105 sb = sb_reference (device);
910106 if (sb == NULL)
910107 {
910108     errset (ENODEV); // No such device.
910109     return (NULL);
910110 }
910111 //
910112 // The super block is available, but the inode is
910113 // not yet cached.
910114 // Verify if the inode map reports it as allocated.
910115 //
910116 status = sb_inode_status (sb, ino);
910117 if (!status)
910118 {
910119     //
910120     // The inode is not allocated and cannot be
910121     // loaded.
910122     //
910123     errset (ENOENT); // No such file or directory.
910124     return (NULL);
```



```
910125     }
910126     //
910127     // The inode was not already cached, but is
910128     // considered as allocated
910129     // inside the inode map. Find a free slot to load
910130     // the inode inside
910131     // the inode table (in memory).
910132     //
910133     inode = inode_reference ((dev_t) - 1, (ino_t) - 1);
910134     if (inode == NULL)
910135     {
910136         errset (ENFILE); // Too many files open in
910137         // system.
910138         return (NULL);
910139     }
910140     //
910141     // A free inode slot was found. The inode must be
910142     // loaded.
910143     // Calculate the memory inode size, to be saved
910144     // inside the file
910145     // system: the administrative inode data, as it is
910146     // saved inside
910147     // the file system. The 'inode_t' type is bigger
910148     // than the real
910149     // inode administrative size, because it contains
910150     // more data, that is
910151     // not saved on disk.
910152     //
910153     size = offsetof (inode_t, sb);
910154     //
910155     // Calculating start position for read.
910156     //
910157     // [1] Boot block.
910158     // [2] Super block.
910159     // [3] Inode bit map.
910160     // [4] Zone bit map.
910161     // [5] Previous inodes: consider that the inode zero
```

```
910162 // is
910163 // present in the inode map, but not in the inode
910164 // table.
910165 //
910166 start = 1024; // [1]
910167 start += 1024; // [2]
910168 start += (sb->map_inode_blocks * 1024); // [3]
910169 start += (sb->map_zone_blocks * 1024); // [4]
910170 start += ((ino - 1) * size); // [5]
910171 //
910172 // Read inode from disk.
910173 //
910174 n =
910175     dev_io ((pid_t) - 1, device, DEV_READ, start,
910176           inode, size, NULL);
910177 if (n != size)
910178     {
910179         errset (EIO); // I/O error.
910180         return (NULL);
910181     }
910182 //
910183 // The inode was read: add some data to the working
910184 // copy in memory.
910185 //
910186 inode->sb = sb;
910187 inode->sb_attached = NULL;
910188 inode->ino = ino;
910189 inode->references = 1;
910190 inode->changed = 0;
910191 //
910192 inode->blkcnt = inode->size;
910193 inode->blkcnt /= sb->blksize;
910194 if (inode->size % sb->blksize)
910195     {
910196         inode->blkcnt++;
910197     }
910198 //
```

```
910199     inode->pipe_dir = 1;  // Pipes must start with
910200     // write.
910201     inode->pipe_off_read = 0;
910202     inode->pipe_off_write = 0;
910203     inode->pipe_ref_read = 0;
910204     inode->pipe_ref_write = 0;
910205     //
910206     // Return the inode pointer.
910207     //
910208     return (inode);
910209 }
```

94.5.17 kernel/fs/inode_pipe_make.c

Si veda la sezione [93.6.16](#).

```
920001 #include <kernel/fs.h>
920002 #include <sys/stat.h>
920003 #include <errno.h>
920004 #include <kernel/lib_k.h>
920005 #include <kernel/lib_s.h>
920006 //-----
920007 inode_t *
920008 inode_pipe_make (void)
920009 {
920010     inode_t *inode;
920011     //
920012     // Find a free inode.
920013     //
920014     inode = inode_reference ((dev_t) - 1, (ino_t) - 1);
920015     if (inode == NULL)
920016     {
920017         //
920018         // No free slot available.
920019         //
920020         errset (ENFILE); // Too many files open in
920021         // system.
```

```
920022     return (NULL);
920023     }
920024     //
920025     // Put data inside the inode. Please note that
920026     // 'inode->ino' must be
920027     // zero, because it is necessary to recognize it as
920028     // an internal
920029     // inode with no file system. Otherwise, with a
920030     // value different than
920031     // zero, 'inode_put()' will try to remove it. [*]
920032     //
920033     inode->mode = S_IFIFO;
920034     inode->uid = 0;
920035     inode->gid = 0;
920036     inode->size = 0;
920037     inode->time = 0;
920038     inode->links = 0;
920039     inode->direct[0] = 0;
920040     inode->direct[1] = 0;
920041     inode->direct[2] = 0;
920042     inode->direct[3] = 0;
920043     inode->direct[4] = 0;
920044     inode->direct[5] = 0;
920045     inode->direct[6] = 0;
920046     inode->indirect1 = 0;
920047     inode->indirect2 = 0;
920048     inode->sb_attached = NULL;
920049     inode->sb = 0;
920050     inode->ino = 0;           // Must be zero. [*]
920051     inode->blkcnt = 0;
920052     inode->references = 1;
920053     inode->changed = 0;
920054     inode->pipe_dir = 1;    // Must start with write.
920055     inode->pipe_off_read = 0;
920056     inode->pipe_off_write = 0;
920057     inode->pipe_ref_read = 0;
920058     inode->pipe_ref_write = 0;
```

```
920059 //
920060 // Add all access permissions.
920061 //
920062 inode->mode |= (S_IRWXU | S_IRWXG | S_IRWXO);
920063 //
920064 // Return the inode pointer.
920065 //
920066 return (inode);
920067 }
```

94.5.18 kernel/fs/inode_pipe_read.c



Si veda la sezione [93.6.17](#).

```
930001 #include <kernel/fs.h>
930002 #include <errno.h>
930003 //-----
930004 ssize_t
930005 inode_pipe_read (inode_t * inode, void *buffer,
930006                 size_t count, int *eof)
930007 {
930008     unsigned char *buffer_s;
930009     unsigned char *buffer_d = buffer;
930010     int i;
930011     //
930012     // The inode pointer must be valid.
930013     //
930014     if (inode == NULL)
930015     {
930016         errset (EINVAL); // Invalid argument.
930017         return ((ssize_t) - 1);
930018     }
930019     //
930020     // Check the current pipe direction and see if can
930021     // be
930022     // read something.
930023     //
```

```
930024     if (inode->pipe_dir)
930025     {
930026         //
930027         // Write: if indexes are the same, cannot read
930028         // anything.
930029         //
930030         if (inode->pipe_off_write == inode->pipe_off_read)
930031         {
930032             //
930033             // Cannot read.
930034             //
930035             if (inode->pipe_ref_write == 0)
930036             {
930037                 if (eof != NULL)
930038                 {
930039                     *eof = 1;
930040                 }
930041             }
930042             return ((ssize_t) 0);
930043         }
930044     }
930045     else
930046     {
930047         //
930048         // Read: the pipe is waiting for a read.
930049         //
930050         ;
930051     }
930052     //
930053     // Might read something. Set the pointer to the
930054     // source buffer,
930055     // that is the area used for direct zones, including
930056     // first
930057     // indirect pointers (total: (7+2)*2 = 18 bytes).
930058     //
930059     buffer_s = (void *) &(inode->direct[0]);
930060     //
```

```
930061     i = 0;
930062     //
930063     if (inode->pipe_off_read >= inode->pipe_off_write)
930064     {
930065         for (; i < count; i++)
930066         {
930067             if (inode->pipe_off_read < INODE_PIPE_BUFFER_SIZE)
930068             {
930069                 buffer_d[i] = buffer_s[inode->pipe_off_read];
930070                 inode->pipe_off_read++;
930071             }
930072             else
930073             {
930074                 inode->pipe_off_read = 0;
930075                 break;
930076             }
930077         }
930078     }
930079     //
930080     if (inode->pipe_off_read < inode->pipe_off_write)
930081     {
930082         for (; i < count; i++)
930083         {
930084             if (inode->pipe_off_read < inode->pipe_off_write)
930085             {
930086                 buffer_d[i] = buffer_s[inode->pipe_off_read];
930087                 inode->pipe_off_read++;
930088             }
930089             else
930090             {
930091                 break;
930092             }
930093         }
930094     }
930095     //
930096     // At this point, it is time to set the direction to
930097     // write;
```

```
930098 // it doesn't matter if the direction is already set
930099 // so.
930100 //
930101 if (inode->pipe_off_read == inode->pipe_off_write)
930102     {
930103         inode->pipe_dir = 1;
930104     }
930105 //
930106 // Ok.
930107 //
930108 return ((ssize_t) i);
930109 }
```

94.5.19 kernel/fs/inode_pipe_write.c



Si veda la sezione [93.6.18](#).

```
940001 #include <kernel/fs.h>
940002 #include <errno.h>
940003 //-----
940004 ssize_t
940005 inode_pipe_write (inode_t * inode, const void *buffer,
940006                  size_t count)
940007 {
940008     const unsigned char *buffer_s = buffer;
940009     unsigned char *buffer_d;
940010     int i;
940011     //
940012     // The inode pointer must be valid.
940013     //
940014     if (inode == NULL)
940015     {
940016         errset (EINVAL); // Invalid argument.
940017         return ((ssize_t) - 1);
940018     }
940019     //
940020     // Check the current pipe direction and see if can
```



```
940021 // be
940022 // written something.
940023 //
940024 if (inode->pipe_dir)
940025 {
940026 //
940027 // Write: the pipe is waiting for a write.
940028 //
940029 ;
940030 }
940031 else
940032 {
940033 //
940034 // Read: if indexes are the same, cannot write
940035 // anything.
940036 //
940037 if (inode->pipe_off_write == inode->pipe_off_read)
940038 {
940039 //
940040 // Cannot write. More checks will be made by
940041 // 's_write()'.
940042 //
940043 return ((ssize_t) 0);
940044 }
940045 }
940046 //
940047 // Might write something. Set the pointer to the
940048 // destination buffer,
940049 // that is the area used for direct zones, including
940050 // first indirect
940051 // pointers (total: (7+2)*2 = 18 bytes).
940052 //
940053 buffer_d = (void *) &(inode->direct[0]);
940054 //
940055 i = 0;
940056 //
940057 if (inode->pipe_off_write >= inode->pipe_off_read)
```

```
940058     {
940059         for (; i < count; i++)
940060             {
940061                 if (inode->pipe_off_write <
940062                     INODE_PIPE_BUFFER_SIZE)
940063                     {
940064                         buffer_d[inode->pipe_off_write] = buffer_s[i];
940065                         inode->pipe_off_write++;
940066                     }
940067                 else
940068                     {
940069                         inode->pipe_off_write = 0;
940070                         break;
940071                     }
940072             }
940073     }
940074     //
940075     if (inode->pipe_off_write < inode->pipe_off_read)
940076     {
940077         for (; i < count; i++)
940078             {
940079                 if (inode->pipe_off_write < inode->pipe_off_read)
940080                     {
940081                         buffer_d[inode->pipe_off_write] = buffer_s[i];
940082                         inode->pipe_off_write++;
940083                     }
940084                 else
940085                     {
940086                         break;
940087                     }
940088             }
940089     }
940090     //
940091     // At this point, it is time to set the direction to
940092     // read;
940093     // it doesn't matter if the direction is already set
940094     // so.
```

```

940095 //
940096 if (inode->pipe_off_write == inode->pipe_off_read)
940097     {
940098         inode->pipe_dir = 0;
940099     }
940100 //
940101 // Ok.
940102 //
940103 return ((ssize_t) i);
940104 }

```

94.5.20 kernel/fs/inode_print.c

Si veda la sezione [93.6.19](#).



```

950001 #include <sys/os32.h>
950002 #include <kernel/fs.h>
950003 #include <kernel/lib_k.h>
950004 #include <time.h>
950005 //-----
950006 void
950007 inode_print (void)
950008 {
950009     int i;
950010     dev_t device_attached = 0;
950011     time_t time;
950012     struct tm *timeptr;
950013     char type;
950014     dev_t device;
950015     //
950016     k_printf
950017         (" dev    ino ref c mntd t mode  uid gid size Kib "
950018          "date      time      lnk dirct[0]\n");
950019     //
950020     for (i = 0; i < INODE_MAX_SLOTS; i++)
950021     {
950022         if (inode_table[i].references <= 0)

```

```
950023     {
950024         continue;
950025     }
950026     //
950027     // Calculate modification time.
950028     //
950029     time = inode_table[i].time;
950030     //
950031     timeptr = gmtime (&time);
950032     //
950033     // Get type from mode.
950034     //
950035     if (S_ISBLK (inode_table[i].mode))
950036         type = 'b';
950037     else if (S_ISCHR (inode_table[i].mode))
950038         type = 'c';
950039     else if (S_ISFIFO (inode_table[i].mode))
950040         type = 'p';
950041     else if (S_ISREG (inode_table[i].mode))
950042         type = '-';
950043     else if (S_ISDIR (inode_table[i].mode))
950044         type = 'd';
950045     else if (S_ISLNK (inode_table[i].mode))
950046         type = 'l';
950047     else if (S_ISSOCK (inode_table[i].mode))
950048         type = 's';
950049     else
950050         type = '?';
950051     //
950052     // Is it a mount point?
950053     //
950054     if (inode_table[i].sb_attached != NULL)
950055     {
950056         device_attached =
950057             inode_table[i].sb_attached->device;
950058     }
950059     //
```

```
950060 // Is there a super block device?
950061 //
950062 if (inode_table[i].sb == NULL)
950063 {
950064     device = 0;
950065 }
950066 else
950067 {
950068     device = inode_table[i].sb->device;
950069 }
950070 //
950071 // Print data.
950072 //
950073 k_printf
950074 ("%04x %5i %3i %c %04x %c %04o %4i %3i %8i "
950075 "%4i.%02i.%02i %2i:%02i:%02i %3i %08x\n",
950076 (unsigned int) device,
950077 (unsigned int) inode_table[i].ino,
950078 (unsigned int) inode_table[i].references,
950079 (inode_table[i].changed ? '!' : ' '),
950080 (unsigned int) device_attached, type,
950081 (unsigned int) inode_table[i].mode,
950082 (unsigned int) inode_table[i].uid,
950083 (unsigned int) inode_table[i].gid,
950084 (unsigned int) (inode_table[i].size / 1024),
950085 timeptr->tm_year, timeptr->tm_mon,
950086 timeptr->tm_mday, timeptr->tm_hour,
950087 timeptr->tm_min, timeptr->tm_sec,
950088 (unsigned int) inode_table[i].links,
950089 (unsigned int) inode_table[i].direct[0]);
950090 }
950091 }
```

94.5.21 kernel/fs/inode_put.c

<<

Si veda la sezione [93.6.20](#).

```
960001 #include <kernel/fs.h>
960002 #include <errno.h>
960003 #include <kernel/lib_k.h>
960004 //-----
960005 int
960006 inode_put (inode_t * inode)
960007 {
960008     int status;
960009     //
960010     // Check for valid argument.
960011     //
960012     if (inode == NULL)
960013     {
960014         errset (EINVAL); // Invalid argument.
960015         return (-1);
960016     }
960017     //
960018     // Check for valid references.
960019     //
960020     if (inode->references <= 0)
960021     {
960022         errset (EUNKNOWN); // Cannot put an inode with
960023         return (-1); // zero or negative
960024         // references.
960025     }
960026     //
960027     // Debug.
960028     //
960029     if (inode->ino != 0 && inode->sb->device == 0)
960030     {
960031         k_printf
960032         ("kernel alert: trying to close "
960033         "inode with device "
960034         "zero, but a number different than zero!\n");
```

```
960035     errset (EUNKNOWN);           // Cannot put an inode
960036     // with
960037     return (-1);           // zero or negative
960038     // references.
960039     }
960040     //
960041     // There is at least one reference: now the
960042     // references value is
960043     // reduced.
960044     //
960045     inode->references--;
960046     inode->changed = 1;
960047     //
960048     // If 'inode->ino' is zero, it means that the inode
960049     // was created in memory, but there is no file system
960050     // for it. For example, it might be a standard I/O
960051     // inode create automatically for a process.
960052     // Inodes with number zero cannot be removed from a
960053     // file system.
960054     //
960055     if (inode->ino == 0)
960056     {
960057         //
960058         // Nothing to do: just return.
960059         //
960060         return (0);
960061     }
960062     //
960063     // References counter might be zero.
960064     //
960065     if (inode->references == 0)
960066     {
960067         //
960068         // Check if the inode is to be deleted (until
960069         // there are
960070         // run time references, the inode cannot be
960071         // removed).
```

```

960072 //
960073 if (inode->links == 0
960074     || (S_ISDIR (inode->mode) && inode->links == 1))
960075 {
960076     //
960077     // The inode has no more run time references
960078     // and file system
960079     // links are also zero (or one for a
960080     // directory): remove it!
960081     //
960082     status = inode_truncate (inode);
960083     if (status != 0)
960084     {
960085         k_perror (NULL);
960086     }
960087     //
960088     inode_free (inode);
960089     return (0);
960090 }
960091 }
960092 //
960093 // Save inode to disk and return.
960094 //
960095 return (inode_save (inode));
960096 }

```

94.5.22 kernel/fs/inode_reference.c

«

Si veda la sezione [93.6.21](#).

```

970001 #include <kernel/fs.h>
970002 #include <errno.h>
970003 //-----
970004 inode_t *
970005 inode_reference (dev_t device, ino_t ino)
970006 {
970007     int s;           // Slot index.

```



```
970008     sb_t *sb_table = sb_reference (0);
970009     //
970010     // If device is zero, and inode is zero, a reference
970011     // to the whole
970012     // table is returned.
970013     //
970014     if (device == 0 && ino == 0)
970015     {
970016         return (inode_table);
970017     }
970018     //
970019     // If device is ((dev_t) -1) and the inode is
970020     // ((ino_t) -1), a
970021     // reference to a free inode slot is returned.
970022     //
970023     if (device == (dev_t) - 1 && ino == ((ino_t) - 1))
970024     {
970025         for (s = 0; s < INODE_MAX_SLOTS; s++)
970026         {
970027             if (inode_table[s].references == 0)
970028             {
970029                 return (&inode_table[s]);
970030             }
970031         }
970032         return (NULL);
970033     }
970034     //
970035     // If device is zero and the inode is 1, a reference
970036     // to the root
970037     // directory inode is returned.
970038     //
970039     if (device == 0 && ino == 1)
970040     {
970041         //
970042         // The super block table is to be scanned.
970043         //
970044         for (device = 0, s = 0; s < SB_MAX_SLOTS; s++)
```

```
970045     {
970046         if (sb_table[s].device != 0
970047             && (sb_table[s].inode_mounted_on->
970048                 sb_attached->device ==
970049                 sb_table[s].device))
970050             {
970051                 device = sb_table[s].device;
970052                 break;
970053             }
970054     }
970055     if (device == 0)
970056     {
970057         errset (E_CANNOT_FIND_ROOT_DEVICE);
970058         return (NULL);
970059     }
970060     //
970061     // Scan the inode table to find inode 1 and the
970062     // same device.
970063     //
970064     for (s = 0; s < INODE_MAX_SLOTS; s++)
970065     {
970066         if (inode_table[s].sb->device == device &&
970067             inode_table[s].ino == 1)
970068             {
970069                 return (&inode_table[s]);
970070             }
970071     }
970072     //
970073     // Cannot find a root file system inode.
970074     //
970075     errset (E_CANNOT_FIND_ROOT_INODE);
970076     return (NULL);
970077 }
970078 //
970079 // A device and an inode number were selected: find
970080 // the inode
970081 // associated to it.
```

```
970082 //
970083 for (s = 0; s < INODE_MAX_SLOTS; s++)
970084 {
970085     if (inode_table[s].sb->device == device &&
970086         inode_table[s].ino == ino)
970087     {
970088         return (&inode_table[s]);
970089     }
970090 }
970091 //
970092 // The inode was not found.
970093 //
970094 return (NULL);
970095 }
```

94.5.23 kernel/fs/inode_save.c

Si veda la sezione [93.6.22](#).

```
980001 #include <kernel/fs.h>
980002 #include <errno.h>
980003 #include <kernel/dev.h>
980004 //-----
980005 int
980006 inode_save (inode_t * inode)
980007 {
980008     size_t size;
980009     unsigned long int start;
980010     ssize_t n;
980011 //
980012 // Check for valid argument.
980013 //
980014 if (inode == NULL)
980015 {
980016     errset (EINVAL); // Invalid argument.
980017     return (-1);
980018 }
```

```
980019 //
980020 // If the inode number is zero, no file system is
980021 // involved!
980022 //
980023 if (inode->ino == 0)
980024 {
980025     return (0);
980026 }
980027 //
980028 // Save the super block to disk.
980029 //
980030 sb_save (inode->sb);
980031 //
980032 // Save the inode to disk.
980033 //
980034 if (inode->changed)
980035 {
980036     size = offsetof (inode_t, sb);
980037     //
980038     // Calculating start position for write.
980039     //
980040     //
980041     // Boot block: 1024 bytes
980042     //
980043     start = 1024;
980044     //
980045     // Super block: + 1024 bytes
980046     //
980047     start += 1024;
980048     //
980049     // Inode bit map:
980050     //
980051     start += (inode->sb->map_inode_blocks * 1024);
980052     //
980053     // Zone bit map:
980054     //
980055     start += (inode->sb->map_zone_blocks * 1024);
```

```

980056      //
980057      // Previous inodes: consider that the inode zero
980058      // is present in the inode map, but not in the
980059      // inode table.
980060      //
980061      start += ((inode->ino - 1) * size);
980062      //
980063      // Write the inode.
980064      //
980065      n =
980066          dev_io ((pid_t) - 1, inode->sb->device,
980067                DEV_WRITE, start, inode, size, NULL);
980068      //
980069      inode->changed = 0;
980070  }
980071  return (0);
980072  }

```

94.5.24 kernel/fs/inode_stdio_dev_make.c



Si veda la sezione [93.6.23](#).

```

990001  #include <kernel/fs.h>
990002  #include <errno.h>
990003  #include <kernel/lib_k.h>
990004  #include <kernel/lib_s.h>
990005  //-----
990006  inode_t *
990007  inode_stdio_dev_make (dev_t device, mode_t mode)
990008  {
990009      inode_t *inode;
990010      //
990011      // Check for arguments.
990012      //
990013      if (mode == 0 || device == 0)
990014          {
990015              errset (EINVAL); // Invalid argument.

```

```
990016     return (NULL);
990017 }
990018 //
990019 // Find a free inode.
990020 //
990021 inode = inode_reference ((dev_t) - 1, (ino_t) - 1);
990022 if (inode == NULL)
990023 {
990024     //
990025     // No free slot available.
990026     //
990027     errset (ENFILE); // Too many files open in
990028     // system.
990029     return (NULL);
990030 }
990031 //
990032 // Put data inside the inode. Please note that
990033 // 'inode->ino' must be
990034 // zero, because it is necessary to recognize it as
990035 // an internal
990036 // inode with no file system. Otherwise, with a
990037 // value different than
990038 // zero, 'inode_put()' will try to remove it. [*]
990039 //
990040 inode->mode = mode;
990041 inode->uid = 0;
990042 inode->gid = 0;
990043 inode->size = 0;
990044 inode->time = s_time ((pid_t) 0, NULL);
990045 inode->links = 0;
990046 inode->direct[0] = device;
990047 inode->direct[1] = 0;
990048 inode->direct[2] = 0;
990049 inode->direct[3] = 0;
990050 inode->direct[4] = 0;
990051 inode->direct[5] = 0;
990052 inode->direct[6] = 0;
```

```

990053     inode->indirect1 = 0;
990054     inode->indirect2 = 0;
990055     inode->sb_attached = NULL;
990056     inode->sb = 0;
990057     inode->ino = 0;          // Must be zero. [*]
990058     inode->blkcnt = 0;
990059     inode->references = 1;
990060     inode->changed = 0;
990061     //
990062     // Add all access permissions.
990063     //
990064     inode->mode |= (S_IRWXU | S_IRWXG | S_IRWXO);
990065     //
990066     // Return the inode pointer.
990067     //
990068     return (inode);
990069 }

```

94.5.25 kernel/fs/inode_truncate.c

Si veda la sezione [93.6.24](#).

```

1000001 #include <kernel/fs.h>
1000002 #include <errno.h>
1000003 #include <kernel/lib_k.h>
1000004 //-----
1000005 int
1000006 inode_truncate (inode_t * inode)
1000007 {
1000008     unsigned int indirect_zones;
1000009     zno_t zone_table1[INODE_MAX_INDIRECT_ZONES];
1000010     zno_t zone_table2[INODE_MAX_INDIRECT_ZONES];
1000011     unsigned int i;          // Direct index.
1000012     unsigned int i0;        // Single indirect index.
1000013     unsigned int i1;        // Double indirect first
1000014     // index.
1000015     unsigned int i2;        // Double indirect second

```

```
1000016 // index.
1000017 int status; // 'zone_read()' return value.
1000018 //
1000019 // Check argument.
1000020 //
1000021 if (inode == NULL)
1000022 {
1000023     errset (EINVAL);
1000024     return (-1);
1000025 }
1000026 //
1000027 // Calculate how many indirect zone numbers are
1000028 // stored inside
1000029 // a zone: it depends on the zone size.
1000030 //
1000031 indirect_zones = inode->sb->blksize / 2;
1000032 //
1000033 // Scan and release direct zones. Errors are
1000034 // ignored.
1000035 //
1000036 for (i = 0; i < 7; i++)
1000037 {
1000038     zone_free (inode->sb, inode->direct[i]);
1000039     inode->direct[i] = 0;
1000040 }
1000041 //
1000042 // Scan single indirect zones, if present.
1000043 //
1000044 if (inode->blkcnt > 7 && inode->indirect1 != 0)
1000045 {
1000046     //
1000047     // There is a single indirect table to load.
1000048     // Errors are
1000049     // almost ignored.
1000050     //
1000051     status =
1000052         zone_read (inode->sb, inode->indirect1,
```



```
100053         zone_table1);
100054     if (status == 0)
100055     {
100056         //
100057         // Scan the table and remove zones.
100058         //
100059         for (i0 = 0; i0 < indirect_zones; i0++)
100060         {
100061             zone_free (inode->sb, zone_table1[i0]);
100062         }
100063     }
100064     //
100065     // Remove indirect table too.
100066     //
100067     zone_free (inode->sb, inode->indirect1);
100068     //
100069     // Clear single indirect reference inside the
100070     // inode.
100071     //
100072     inode->indirect1 = 0;
100073 }
100074 //
100075 // Scan double indirect zones, if present.
100076 //
100077 if (inode->blkcnt > (7 + indirect_zones)
100078     && inode->indirect2 != 0)
100079 {
100080     //
100081     // There is a double indirect table to load.
100082     // Errors are
100083     // almost ignored.
100084     //
100085     status =
100086         zone_read (inode->sb, inode->indirect2,
100087                 zone_table1);
100088     if (status == 0)
100089     {
```

```
1000090 //
1000091 // Scan the table and get second level
1000092 // indirection.
1000093 //
1000094 for (i1 = 0; i1 < indirect_zones; i1++)
1000095 {
1000096     if ((inode->blkcnt
1000097         >
1000098         (7 + indirect_zones +
1000099         indirect_zones * i1))
1000100     && zone_table1[i1] != 0)
1000101     {
1000102         //
1000103         // There is a second level table to
1000104         // load.
1000105         //
1000106         status =
1000107             zone_read (inode->sb,
1000108                       zone_table1[i1],
1000109                       zone_table2);
1000110         if (status == 0)
1000111         {
1000112             //
1000113             // Release zones.
1000114             //
1000115             for (i2 = 0;
1000116                 i2 < indirect_zones &&
1000117                 (inode->blkcnt >
1000118                 (7 + indirect_zones +
1000119                 indirect_zones * i1 +
1000120                 i2)); i2++)
1000121                 {
1000122                     zone_free (inode->sb,
1000123                                 zone_table2[i2]);
1000124                 }
1000125             //
1000126             // Remove second level indirect
```

```

1000127         // table.
1000128         //
1000129         zone_free (inode->sb,
1000130                   zone_table1[i1]);
1000131     }
1000132 }
1000133 }
1000134 //
1000135 // Remove first level indirect table.
1000136 //
1000137 zone_free (inode->sb, inode->indirect2);
1000138 }
1000139 //
1000140 // Clear single indirect reference inside the
1000141 // inode.
1000142 //
1000143 inode->indirect2 = 0;
1000144 }
1000145 //
1000146 // Update super block and inode data.
1000147 //
1000148 sb_save (inode->sb);
1000149 inode->size = 0;
1000150 inode->changed = 1;
1000151 inode_save (inode);
1000152 //
1000153 // Successful return.
1000154 //
1000155 return (0);
1000156 }

```

94.5.26 kernel/fs/inode_zone.c

Si veda la sezione [93.6.25](#).

```

1010001 #include <kernel/fs.h>
1010002 #include <errno.h>

```

```
1010003 #include <kernel/lib_k.h>
1010004 //-----
1010005 zno_t
1010006 inode_zone (inode_t * inode, zno_t fzone, int write)
1010007 {
1010008     unsigned int indirect_zones;
1010009     unsigned int allocated_zone;
1010010     zno_t zone_table[INODE_MAX_INDIRECT_ZONES];
1010011     char buffer[SB_MAX_ZONE_SIZE];
1010012     unsigned int i0;      // Single indirect index.
1010013     unsigned int i1;      // Double indirect first
1010014     // index.
1010015     unsigned int i2;      // Double indirect second
1010016     // index.
1010017     int status;
1010018     zno_t zone_second;    // Second level table zone.
1010019     //
1010020     // Check to have a valid inode.
1010021     //
1010022     if (inode == NULL)
1010023     {
1010024         errset (EINVAL);
1010025         return ((zno_t) - 1);
1010026     }
1010027     //
1010028     // Calculate how many indirect zone numbers are
1010029     // stored inside
1010030     // a zone: it depends on the zone size.
1010031     //
1010032     indirect_zones = inode->sb->blksize / 2;
1010033     //
1010034     // Convert file-zone number into a zone number.
1010035     //
1010036     if (fzone < 7)
1010037     {
1010038         //
1010039         // 0 <= fzone <= 6
```

```
1010040 // The zone number is inside the direct zone
1010041 // references.
1010042 // Verify to have such zone.
1010043 //
1010044 if (inode->direct[fzone] == 0)
1010045 {
1010046 //
1010047 // There is not such zone, but we do not
1010048 // consider
1010049 // it an error, because a file can be not
1010050 // contiguous.
1010051 //
1010052 if (!write)
1010053 {
1010054     return ((zno_t) 0);
1010055 }
1010056 //
1010057 // Must be allocated.
1010058 //
1010059 allocated_zone = zone_alloc (inode->sb);
1010060 if (allocated_zone == 0)
1010061 {
1010062 //
1010063 // Cannot allocate the zone. The
1010064 // variable 'errno' is
1010065 // set by 'zone_alloc()'.
1010066 //
1010067     return ((zno_t) - 1);
1010068 }
1010069 //
1010070 // The zone is allocated: clear the zone and
1010071 // save.
1010072 //
1010073 memset (buffer, 0, SB_MAX_ZONE_SIZE);
1010074 status =
1010075     zone_write (inode->sb, allocated_zone, buffer);
1010076 if (status < 0)
```

```
1010077         {
1010078             //
1010079             // Cannot overwrite the zone. The
1010080             // variable 'errno' is
1010081             // set by 'zone_write()'.
1010082             //
1010083             return ((zno_t) - 1);
1010084         }
1010085         //
1010086         // The zone is allocated and cleared: save
1010087         // the inode.
1010088         //
1010089         inode->direct[fzone] = allocated_zone;
1010090         inode->changed = 1;
1010091         status = inode_save (inode);
1010092         if (status != 0)
1010093             {
1010094                 //
1010095                 // Cannot save the inode. The variable
1010096                 // 'errno' is
1010097                 // set 'inode_save()'.
1010098                 //
1010099                 return ((zno_t) - 1);
1010100             }
1010101         }
1010102         //
1010103         // The zone is there: return it.
1010104         //
1010105         return (inode->direct[fzone]);
1010106     }
1010107     if (fzone < 7 + indirect_zones)
1010108     {
1010109         //
1010110         // 7 <= fzone <= (6 + indirect_zones)
1010111         // The zone number is inside the single indirect
1010112         // zone
1010113         // references: verify to have the indirect zone
```

```
1010114 // table.
1010115 //
1010116 if (inode->indirect1 == 0)
1010117 {
1010118     //
1010119     // There is not such zone, but it is not an
1010120     // error.
1010121     //
1010122     if (!write)
1010123     {
1010124         return ((zno_t) 0);
1010125     }
1010126     //
1010127     // The first level of indirection must be
1010128     // initialized.
1010129     //
1010130     allocated_zone = zone_alloc (inode->sb);
1010131     if (allocated_zone == 0)
1010132     {
1010133         //
1010134         // Cannot allocate the zone for the
1010135         // indirection table:
1010136         // this is an error and the 'errno'
1010137         // value is produced
1010138         // by 'zone_alloc()'.
1010139         //
1010140         return ((zno_t) - 1);
1010141     }
1010142     //
1010143     // The zone for the indirection table is
1010144     // allocated:
1010145     // clear the zone and save.
1010146     //
1010147     memset (buffer, 0, SB_MAX_ZONE_SIZE);
1010148     status =
1010149         zone_write (inode->sb, allocated_zone, buffer);
1010150     if (status < 0)
```

```
1010151         {
1010152             //
1010153             // Cannot overwrite the zone. The
1010154             // variable 'errno' is
1010155             // set by 'zone_write()'.
1010156             //
1010157             return ((zno_t) - 1);
1010158         }
1010159         //
1010160         // The indirection table zone is allocated
1010161         // and cleared:
1010162         // save the inode.
1010163         //
1010164         inode->indirect1 = allocated_zone;
1010165         inode->changed = 1;
1010166         status = inode_save (inode);
1010167         if (status != 0)
1010168             {
1010169                 //
1010170                 // Cannot save the inode. This is an
1010171                 // error and the value
1010172                 // for 'errno' is produced by
1010173                 // 'inode_save()'.
1010174                 //
1010175                 return ((zno_t) - 1);
1010176             }
1010177     }
1010178     //
1010179     // An indirect table is present inside the file
1010180     // system:
1010181     // load it.
1010182     //
1010183     status =
1010184         zone_read (inode->sb, inode->indirect1, zone_table);
1010185     if (status != 0)
1010186         {
1010187             //
```



```
1010188      // Cannot load the indirect table. This is
1010189      // an error and the
1010190      // value for 'errno' is assigned by function
1010191      // 'zone_read()'.
1010192      //
1010193      return ((zno_t) - 1);
1010194    }
1010195    //
1010196    // The indirect table was read. Calculate the
1010197    // index inside
1010198    // the table, for the requested zone.
1010199    //
1010200    i0 = (fzone - 7);
1010201    //
1010202    // Check if the zone is to be allocated.
1010203    //
1010204    if (zone_table[i0] == 0)
1010205    {
1010206        //
1010207        // There is not such zone, but it is not an
1010208        // error.
1010209        //
1010210        if (!write)
1010211        {
1010212            return ((zno_t) 0);
1010213        }
1010214        //
1010215        // The zone must be allocated.
1010216        //
1010217        allocated_zone = zone_alloc (inode->sb);
1010218        if (allocated_zone == 0)
1010219        {
1010220            //
1010221            // There is no space for the zone
1010222            // allocation. The
1010223            // variable 'errno' is already updated
1010224            // by
```

```
1010225         // 'zone_alloc()'.
1010226         //
1010227         return ((zno_t) - 1);
1010228     }
1010229     //
1010230     // The zone is allocated: clear the zone and
1010231     // save.
1010232     //
1010233     memset (buffer, 0, SB_MAX_ZONE_SIZE);
1010234     status =
1010235         zone_write (inode->sb, allocated_zone, buffer);
1010236     if (status < 0)
1010237     {
1010238         //
1010239         // Cannot overwrite the zone. The
1010240         // variable 'errno' is
1010241         // set by 'zone_write()'.
1010242         //
1010243         return ((zno_t) - 1);
1010244     }
1010245     //
1010246     // The zone is allocated and cleared: update
1010247     // the indirect
1010248     // zone table and save it. The inode is not
1010249     // modified,
1010250     // because the indirect table is outside.
1010251     //
1010252     zone_table[i0] = allocated_zone;
1010253     status =
1010254         zone_write (inode->sb, inode->indirect1,
1010255                   zone_table);
1010256     if (status != 0)
1010257     {
1010258         //
1010259         // Cannot save the zone. The variable
1010260         // 'errno' is already
1010261         // set by 'zone_write()'.
```

```
1010262         //
1010263         return ((zno_t) - 1);
1010264     }
1010265 }
1010266 //
1010267 // The zone is allocated.
1010268 //
1010269 return (zone_table[i0]);
1010270 }
1010271 else
1010272 {
1010273     //
1010274     // (7 + indirect_zones) <= fzone
1010275     // The zone number is inside the double indirect
1010276     // zone
1010277     // references.
1010278     // Verify to have the first level of second
1010279     // indirection.
1010280     //
1010281     if (inode->indirect2 == 0)
1010282     {
1010283         //
1010284         // There is not such zone, but it is not an
1010285         // error.
1010286         //
1010287         if (!write)
1010288         {
1010289             return ((zno_t) 0);
1010290         }
1010291         //
1010292         // The first level of second indirection
1010293         // must be
1010294         // initialized.
1010295         //
1010296         allocated_zone = zone_alloc (inode->sb);
1010297         if (allocated_zone == 0)
1010298         {
```

```
1010299          //
1010300          // Cannot allocate the zone. The
1010301          // variable 'errno' is
1010302          // set by 'zone_alloc()'.
1010303          //
1010304          return ((zno_t) - 1);
1010305      }
1010306      //
1010307      // The zone for the indirection table is
1010308      // allocated:
1010309      // clear the zone and save.
1010310      //
1010311      memset (buffer, 0, SB_MAX_ZONE_SIZE);
1010312      status =
1010313          zone_write (inode->sb, allocated_zone, buffer);
1010314      if (status < 0)
1010315          {
1010316              //
1010317              // Cannot overwrite the zone. The
1010318              // variable 'errno' is
1010319              // set by 'zone_write()'.
1010320              //
1010321              return ((zno_t) - 1);
1010322          }
1010323      //
1010324      // The zone for the indirection table is
1010325      // allocated and
1010326      // cleared: save the inode.
1010327      //
1010328      inode->indirect2 = allocated_zone;
1010329      inode->changed = 1;
1010330      status = inode_save (inode);
1010331      if (status != 0)
1010332          {
1010333              //
1010334              // Cannot save the inode. The variable
1010335              // 'errno' is
```

```
1010336         // set by 'inode_save()'.
1010337         //
1010338         return ((zno_t) - 1);
1010339     }
1010340 }
1010341 //
1010342 // The first level of second indirection is
1010343 // present:
1010344 // Read the second indirect table.
1010345 //
1010346 status =
1010347     zone_read (inode->sb, inode->indirect2, zone_table);
1010348 if (status != 0)
1010349     {
1010350         //
1010351         // Cannot read the second indirect table.
1010352         // The variable
1010353         // 'errno' is set by 'zone_read()'.
1010354         //
1010355         return ((zno_t) - 1);
1010356     }
1010357 //
1010358 // The first double indirect table was read:
1010359 // calculate
1010360 // indexes inside first and second level of
1010361 // table.
1010362 //
1010363 fzone -= 7;
1010364 fzone -= indirect_zones;
1010365 i1 = fzone / indirect_zones;
1010366 i2 = fzone % indirect_zones;
1010367 //
1010368 // Verify to have a second level.
1010369 //
1010370 if (zone_table[i1] == 0)
1010371     {
1010372         //
```

```
1010373 // There is not such zone, but it is not an
1010374 // error.
1010375 //
1010376 if (!write)
1010377     {
1010378         return ((zno_t) 0);
1010379     }
1010380 //
1010381 // The second level must be initialized.
1010382 //
1010383 allocated_zone = zone_alloc (inode->sb);
1010384 if (allocated_zone == 0)
1010385     {
1010386         //
1010387         // Cannot allocate the zone. The
1010388         // variable 'errno' is set
1010389         // by 'zone_alloc()'.
1010390         //
1010391         return ((zno_t) - 1);
1010392     }
1010393 //
1010394 // The zone for the indirection table is
1010395 // allocated:
1010396 // clear the zone and save.
1010397 //
1010398 memset (buffer, 0, SB_MAX_ZONE_SIZE);
1010399 status =
1010400     zone_write (inode->sb, allocated_zone, buffer);
1010401 if (status < 0)
1010402     {
1010403         //
1010404         // Cannot overwrite the zone. The
1010405         // variable 'errno' is
1010406         // set by 'zone_write()'.
1010407         //
1010408         return ((zno_t) - 1);
1010409     }
```

```
1010410 //
1010411 // Update the first level index and save it.
1010412 //
1010413 zone_table[i1] = allocated_zone;
1010414 status =
1010415     zone_write (inode->sb, inode->indirect2,
1010416                zone_table);
1010417 if (status != 0)
1010418     {
1010419         //
1010420         // Cannot write the zone. The variable
1010421         // 'errno' is set
1010422         // by 'zone_write()'.
1010423         //
1010424         return ((zno_t) - 1);
1010425     }
1010426 }
1010427 //
1010428 // The second level can be read, overwriting the
1010429 // array
1010430 // 'zone_table[]'. The zone number for the
1010431 // second level
1010432 // indirection table is saved inside
1010433 // 'zone_second', before
1010434 // overwriting the array.
1010435 //
1010436 zone_second = zone_table[i1];
1010437 status =
1010438     zone_read (inode->sb, zone_second, zone_table);
1010439 if (status != 0)
1010440     {
1010441         //
1010442         // Cannot read the second level indirect
1010443         // table. The variable
1010444         // 'errno' is set by 'zone_read()'.
1010445         //
1010446         return ((zno_t) - 1);
```

```
1010447     }
1010448     //
1010449     // The second level was read and 'zone_table[]'
1010450     // is now
1010451     // such second one: check if the zone is to be
1010452     // allocated.
1010453     //
1010454     if (zone_table[i2] == 0)
1010455     {
1010456         //
1010457         // There is not such zone, but it is not an
1010458         // error.
1010459         //
1010460         if (!write)
1010461         {
1010462             return ((zno_t) 0);
1010463         }
1010464         //
1010465         // Must be allocated.
1010466         //
1010467         allocated_zone = zone_alloc (inode->sb);
1010468         if (allocated_zone == 0)
1010469         {
1010470             //
1010471             // Cannot allocate the zone. The
1010472             // variable 'errno' is set
1010473             // by 'zone_alloc()'.
1010474             //
1010475             return ((zno_t) - 1);
1010476         }
1010477         //
1010478         // The zone is allocated: clear the zone and
1010479         // save.
1010480         //
1010481         memset (buffer, 0, SB_MAX_ZONE_SIZE);
1010482         status =
1010483             zone_write (inode->sb, allocated_zone, buffer);
```



```
1010484     if (status < 0)
1010485     {
1010486         //
1010487         // Cannot overwrite the zone. The
1010488         // variable 'errno' is
1010489         // set by 'zone_write()'.
1010490         //
1010491         return ((zno_t) - 1);
1010492     }
1010493     //
1010494     // The zone was allocated and cleared:
1010495     // update the indirect
1010496     // zone table and save it. The inode is not
1010497     // modified, because
1010498     // the indirect table is outside.
1010499     //
1010500     zone_table[i2] = allocated_zone;
1010501     status =
1010502         zone_write (inode->sb, zone_second, zone_table);
1010503     if (status != 0)
1010504     {
1010505         //
1010506         // Cannot write the zone. The variable
1010507         // 'errno' is set
1010508         // by 'zone_write()'.
1010509         //
1010510         return ((zno_t) - 1);
1010511     }
1010512 }
1010513 //
1010514 // The zone is there: return the zone number.
1010515 //
1010516 return (zone_table[i2]);
1010517 }
1010518 }
```

94.5.27 kernel/fs/path_device.c

<<

Si veda la sezione [93.6.38](#).

```
1020001 #include <kernel/fs.h>
1020002 #include <errno.h>
1020003 #include <kernel/proc.h>
1020004 //-----
1020005 dev_t
1020006 path_device (pid_t pid, const char *path)
1020007 {
1020008     proc_t *ps;
1020009     inode_t *inode;
1020010     dev_t device;
1020011     //
1020012     // Get process.
1020013     //
1020014     ps = proc_reference (pid);
1020015     //
1020016     inode = path_inode (pid, path);
1020017     if (inode == NULL)
1020018     {
1020019         errset (errno);
1020020         return ((dev_t) - 1);
1020021     }
1020022     //
1020023     if (!(S_ISBLK (inode->mode) || S_ISCHR (inode->mode)))
1020024     {
1020025         errset (ENODEV); // No such device.
1020026         inode_put (inode);
1020027         return ((dev_t) - 1);
1020028     }
1020029     //
1020030     device = inode->direct[0];
1020031     inode_put (inode);
1020032     return (device);
1020033 }
```

94.5.28 kernel/fs/path_fix.c



Si veda la sezione [93.6.39](#).

```
1030001 #include <kernel/fs.h>
1030002 #include <errno.h>
1030003 #include <kernel/proc.h>
1030004 //-----
1030005 int
1030006 path_fix (char *path)
1030007 {
1030008     char new_path[PATH_MAX];
1030009     char *token[PATH_MAX / 4];
1030010     int t;          // Token index.
1030011     int token_size; // Token array effective size.
1030012     int comp;      // String compare return value.
1030013     size_t path_size; // Path string size.
1030014     //
1030015     // Initialize token search.
1030016     //
1030017     token[0] = strtok (path, "/");
1030018     //
1030019     // Scan tokens.
1030020     //
1030021     for (t = 0;
1030022          t < PATH_MAX / 4 && token[t] != NULL;
1030023          t++, token[t] = strtok (NULL, "/"))
1030024     {
1030025         //
1030026         // If current token is '.', just ignore it.
1030027         //
1030028         comp = strcmp (token[t], ".");
1030029         if (comp == 0)
1030030         {
1030031             t--;
1030032         }
1030033         //
1030034         // If current token is '..', remove previous
```

```
1030035     // token,
1030036     // if there is one.
1030037     //
1030038     comp = strcmp (token[t], "..");
1030039     if (comp == 0)
1030040     {
1030041         if (t > 0)
1030042         {
1030043             t -= 2;
1030044         }
1030045         else
1030046         {
1030047             t = -1;
1030048         }
1030049     }
1030050     //
1030051     // 't' will be incremented and another token
1030052     // will be
1030053     // found.
1030054     //
1030055 }
1030056 //
1030057 // Save the token array effective size.
1030058 //
1030059 token_size = t;
1030060 //
1030061 // Initialize the new path string.
1030062 //
1030063 new_path[0] = '\\0';
1030064 //
1030065 // Build the new path string.
1030066 //
1030067 if (token_size > 0)
1030068 {
1030069     for (t = 0; t < token_size; t++)
1030070     {
1030071         path_size = strlen (new_path);
```

```
1030072         strcat (new_path, "/", 2);
1030073         strcat (new_path, token[t],
1030074                 PATH_MAX - path_size - 1);
1030075     }
1030076 }
1030077 else
1030078 {
1030079     strcat (new_path, "/", 2);
1030080 }
1030081 //
1030082 // Copy the new path into the original string.
1030083 //
1030084 strncpy (path, new_path, PATH_MAX);
1030085 //
1030086 // Return.
1030087 //
1030088 return (0);
1030089 }
```

94.5.29 kernel/fs/path_full.c

Si veda la sezione [93.6.40](#).

```
1040001 #include <kernel/fs.h>
1040002 #include <errno.h>
1040003 #include <kernel/proc.h>
1040004 //-----
1040005 int
1040006 path_full (const char *path, const char *path_cwd,
1040007            char *full_path)
1040008 {
1040009     unsigned int path_size;
1040010     //
1040011     // Check some arguments.
1040012     //
1040013     if (path == NULL || strlen (path) == 0
1040014         || full_path == NULL)
```

```
1040015     {
1040016         errset (EINVAL);  // Invalid argument.
1040017         return (-1);
1040018     }
1040019     //
1040020     // The main path and the receiving one are right.
1040021     // Now arrange to get a full path name.
1040022     //
1040023     if (path[0] == '/')
1040024     {
1040025         strncpy (full_path, path, PATH_MAX);
1040026         full_path[PATH_MAX - 1] = 0;
1040027     }
1040028     else
1040029     {
1040030         if (path_cwd == NULL || strlen (path_cwd) == 0)
1040031         {
1040032             errset (EINVAL);      // Invalid argument.
1040033             return (-1);
1040034         }
1040035         strncpy (full_path, path_cwd, PATH_MAX);
1040036         path_size = strlen (full_path);
1040037         strncat (full_path, "/", (PATH_MAX - path_size));
1040038         path_size = strlen (full_path);
1040039         strncat (full_path, path, (PATH_MAX - path_size));
1040040     }
1040041     //
1040042     // Fix path name so that it has no '..', '.', and no
1040043     // multiple '/'.  
1040044     //
1040045     path_fix (full_path);
1040046     //
1040047     // Return.
1040048     //
1040049     return (0);
1040050 }
```

94.5.30 kernel/fs/path_inode.c



Si veda la sezione [93.6.41](#).

```
1050001 #include <kernel/fs.h>
1050002 #include <errno.h>
1050003 #include <kernel/proc.h>
1050004 #include <kernel/lib_k.h>
1050005 //-----
1050006 #define DIRECTORY_BUFFER_SIZE (SB_MAX_ZONE_SIZE/16)
1050007 //-----
1050008 inode_t *
1050009 path_inode (pid_t pid, const char *path)
1050010 {
1050011     proc_t *ps;
1050012     inode_t *inode;
1050013     dev_t device;
1050014     char full_path[PATH_MAX];
1050015     char *name;
1050016     char *next;
1050017     directory_t dir[DIRECTORY_BUFFER_SIZE];
1050018     char dir_name[NAME_MAX + 1];
1050019     off_t offset_dir;
1050020     ssize_t size_read;
1050021     size_t dir_size_read;
1050022     ssize_t size_to_read;
1050023     int comp;
1050024     int d;           // Directory index;
1050025     int status;     // inode_check() return status.
1050026     //
1050027     // Get process.
1050028     //
1050029     ps = proc_reference (pid);
1050030     //
1050031     // Arrange to get a packed full path name.
1050032     //
1050033     path_full (path, ps->path_cwd, full_path);
1050034     //
```

```
1050035 // Get the root file system inode.
1050036 //
1050037 inode = inode_get ((dev_t) 0, 1);
1050038 if (inode == NULL)
1050039     {
1050040         errset (errno);
1050041         return (NULL);
1050042     }
1050043 //
1050044 // Save the device number.
1050045 //
1050046 device = inode->sb->device;
1050047 //
1050048 // Variable 'inode' already points to the root file
1050049 // system inode:
1050050 // It must be a directory!
1050051 //
1050052 status =
1050053     inode_check (inode, S_IFDIR, 1, ps->euid, ps->egid);
1050054 if (status != 0)
1050055     {
1050056         //
1050057         // Variable 'errno' should be set by
1050058         // inode_check().
1050059         //
1050060         errset (errno);
1050061         inode_put (inode);
1050062         return (NULL);
1050063     }
1050064 //
1050065 // Initialize string scan: find the first path
1050066 // token, after the
1050067 // first '/'.
1050068 //
1050069 name = strtok (full_path, "/");
1050070 //
1050071 // If the original full path is just '/' the
```



```
1050072 // variable 'name'
1050073 // appears as a null pointer, and the variable
1050074 // 'inode' is already
1050075 // what we are looking for.
1050076 //
1050077 if (name == NULL)
1050078     {
1050079     return (inode);
1050080     }
1050081 //
1050082 // There is at least a name after '/' inside the
1050083 // original full
1050084 // path. A scan is going to start: the original
1050085 // value for variable
1050086 // 'inode' is a pointer to the root directory inode.
1050087 //
1050088 for (;;)
1050089     {
1050090     //
1050091     // Find next token.
1050092     //
1050093     next = strtok (NULL, "/");
1050094     //
1050095     // Read the directory from the current inode.
1050096     //
1050097     for (offset_dir = 0;; offset_dir += size_read)
1050098         {
1050099         size_to_read = DIRECTORY_BUFFER_SIZE;
1050100         //
1050101         if ((offset_dir + size_to_read) > inode->size)
1050102             {
1050103             size_to_read = inode->size - offset_dir;
1050104             }
1050105         //
1050106         size_read =
1050107             inode_file_read (inode, offset_dir, dir,
1050108                             size_to_read, NULL);
```

```
1050109 //
1050110 // The size read must be a multiple of 16.
1050111 //
1050112 size_read = ((size_read / 16) * 16);
1050113 //
1050114 // Check anyway if it is zero.
1050115 //
1050116 if (size_read == 0)
1050117 {
1050118 //
1050119 // The directory is ended: release the
1050120 // inode and return.
1050121 //
1050122 inode_put (inode);
1050123 errset (ENOENT); // No such file or
1050124 // directory.
1050125 return (NULL);
1050126 }
1050127 //
1050128 // Calculate how many directory items we
1050129 // have read.
1050130 //
1050131 dir_size_read = size_read / 16;
1050132 //
1050133 // Scan the directory to find the current
1050134 // name.
1050135 //
1050136 for (d = 0; d < dir_size_read; d++)
1050137 {
1050138 //
1050139 // Ensure to have a null terminated
1050140 // string for
1050141 // the name found.
1050142 //
1050143 memcpy (dir_name, dir[d].name,
1050144         (size_t) NAME_MAX);
1050145 dir_name[NAME_MAX] = 0;
```

```
1050146         //
1050147         comp = strcmp (name, dir_name);
1050148         if (comp == 0 && dir[d].ino != 0)
1050149             {
1050150                 //
1050151                 // Found the name and verified that
1050152                 // it has a link to
1050153                 // a inode. Now release the
1050154                 // directory inode.
1050155                 //
1050156                 inode_put (inode);
1050157                 //
1050158                 // Get next inode and break the
1050159                 // loop.
1050160                 //
1050161                 inode = inode_get (device, dir[d].ino);
1050162                 break;
1050163             }
1050164     }
1050165     //
1050166     // If index 'd' is in a valid range, the
1050167     // name was found.
1050168     //
1050169     if (d < dir_size_read)
1050170         {
1050171             //
1050172             // The name was found.
1050173             //
1050174             break;
1050175         }
1050176     }
1050177     //
1050178     // If the function is still working, a file or a
1050179     // directory
1050180     // was found: see if there is another name after
1050181     // this one
1050182     // to look for. If there isn't, just break the
```

```
1050183 // loop.
1050184 //
1050185 if (next == NULL)
1050186 {
1050187     //
1050188     // As no other tokens are to be found, break
1050189     // the loop.
1050190     //
1050191     break;
1050192 }
1050193 //
1050194 // As there is another name after the current
1050195 // one,
1050196 // the current file must be a directory.
1050197 //
1050198 status =
1050199     inode_check (inode, S_IFDIR, 1, ps->euid, ps->egid);
1050200 if (status != 0)
1050201 {
1050202     //
1050203     // Variable 'errno' is set by
1050204     // 'inode_check()'.
1050205     //
1050206     errset (errno);
1050207     inode_put (inode);
1050208     return (NULL);
1050209 }
1050210 //
1050211 // The inode is a directory and the user has the
1050212 // necessary
1050213 // permissions: check if it is a mount point and
1050214 // go to the
1050215 // new device root directory if necessary.
1050216 //
1050217 if (inode->sb_attached != NULL)
1050218 {
1050219     //
```

```
1050220 // Must find the root directory for the new
1050221 // device, and
1050222 // then go to that inode.
1050223 //
1050224 device = inode->sb_attached->device;
1050225 inode_put (inode);
1050226 inode = inode_get (device, 1);
1050227 status = inode_check (inode, S_IFDIR, 1,
1050228                      ps->euid, ps->egid);
1050229 if (status != 0)
1050230     {
1050231         inode_put (inode);
1050232         return (NULL);
1050233     }
1050234 }
1050235 //
1050236 // As a directory was found, and another token
1050237 // follows it,
1050238 // must continue the token scan.
1050239 //
1050240 name = next;
1050241 }
1050242 //
1050243 // Current inode found is the file represented by
1050244 // the requested
1050245 // path.
1050246 //
1050247 return (inode);
1050248 }
```

94.5.31 kernel/fs/path_inode_link.c

Si veda la sezione [93.6.42](#).

```
1060001 #include <kernel/fs.h>
1060002 #include <errno.h>
1060003 #include <kernel/proc.h>
```

```
1060004 #include <libgen.h>
1060005 //-----
1060006 inode_t *
1060007 path_inode_link (pid_t pid, const char *path,
1060008                 inode_t * inode, mode_t mode)
1060009 {
1060010     proc_t *ps;
1060011     char buffer[SB_MAX_ZONE_SIZE];
1060012     off_t start;
1060013     int d;           // Directory index.
1060014     ssize_t size_read;
1060015     ssize_t size_written;
1060016     directory_t *dir = (directory_t *) buffer;
1060017     char path_copy1[PATH_MAX];
1060018     char path_copy2[PATH_MAX];
1060019     char *path_directory;
1060020     char *path_name;
1060021     inode_t *inode_directory;
1060022     inode_t *inode_new;
1060023     dev_t device;
1060024     int status;
1060025     //
1060026     // Check arguments.
1060027     //
1060028     if (path == NULL || strlen (path) == 0)
1060029     {
1060030         errset (EINVAL); // Invalid argument:
1060031         return (NULL); // the path is mandatory.
1060032     }
1060033     //
1060034     if (inode == NULL && mode == 0)
1060035     {
1060036         errset (EINVAL); // Invalid argument: if the
1060037         // inode is to
1060038         return (NULL); // be created, the mode is
1060039         // mandatory.
1060040     }
```

```
1060041 //
1060042 if (inode != NULL)
1060043 {
1060044     if (mode != 0)
1060045     {
1060046         errset (EINVAL);           // Invalid argument:
1060047         // if the inode is
1060048         return (NULL);             // already present,
1060049         // the creation mode
1060050     }                               // must not be given.
1060051     if (S_ISDIR (inode->mode))
1060052     {
1060053         errset (EPERM);           // Operation not
1060054         // permitted.
1060055         return (NULL);           // Refuse to link
1060056         // directory.
1060057     }
1060058     if (inode->links >= LINK_MAX)
1060059     {
1060060         errset (EMLINK);         // Too many links.
1060061         return (NULL);
1060062     }
1060063 }
1060064 //
1060065 // Get process.
1060066 //
1060067 ps = proc_reference (pid);
1060068 //
1060069 // If the destination path already exists, the link
1060070 // cannot be made.
1060071 // It does not matter if the inode is known or not.
1060072 //
1060073 inode_new = path_inode ((uid_t) 0, path);
1060074 if (inode_new != NULL)
1060075 {
1060076     //
1060077     // A file already exists with the same name.
```

```
1060078     //
1060079     inode_put (inode_new);
1060080     errset (EEXIST); // File exists.
1060081     return (NULL);
1060082 }
1060083 //
1060084 // At this point, 'inode_new' is 'NULL'.
1060085 // Copy the source path inside the directory path
1060086 // and name arrays.
1060087 //
1060088     strncpy (path_copy1, path, PATH_MAX);
1060089     strncpy (path_copy2, path, PATH_MAX);
1060090     //
1060091     // Reduce to directory name and find the last name.
1060092     //
1060093     path_directory = dirname (path_copy1);
1060094     path_name = basename (path_copy2);
1060095     if (strlen (path_directory) == 0
1060096         || strlen (path_name) == 0)
1060097     {
1060098         errset (EACCES); // Permission denied: maybe
1060099         // the
1060100         // original path is the root directory
1060101         // and cannot find a previous directory.
1060102         return (NULL);
1060103     }
1060104     //
1060105     // Get the directory inode.
1060106     //
1060107     inode_directory = path_inode (pid, path_directory);
1060108     if (inode_directory == NULL)
1060109     {
1060110         errset (errno);
1060111         return (NULL);
1060112     }
1060113     //
1060114     // Check if something is mounted on it.
```



```
1060115 //
1060116 if (inode_directory->sb_attached != NULL)
1060117 {
1060118     //
1060119     // Must select the right directory.
1060120     //
1060121     device = inode_directory->sb_attached->device;
1060122     inode_put (inode_directory);
1060123     inode_directory = inode_get (device, 1);
1060124     if (inode_directory == NULL)
1060125     {
1060126         return (NULL);
1060127     }
1060128 }
1060129 //
1060130 // If the inode to link is known, check if the
1060131 // selected directory
1060132 // has the same super block than the inode to link.
1060133 //
1060134 if (inode != NULL && inode_directory->sb != inode->sb)
1060135 {
1060136     inode_put (inode_directory);
1060137     errset (ENOENT); // No such file or directory.
1060138     return (NULL);
1060139 }
1060140 //
1060141 // Check if write is allowed for the file system.
1060142 //
1060143 if (inode_directory->sb->options & MOUNT_RO)
1060144 {
1060145     inode_put (inode_directory);
1060146     errset (EROFS); // Read-only file system.
1060147     return (NULL);
1060148 }
1060149 //
1060150 // Verify access permissions for the directory. The
1060151 // number "3" means
```

```
1060152 // that the user must have access permission and
1060153 // write permission:
1060154 // "-wx" == 2+1 == 3.
1060155 //
1060156 status = inode_check (inode_directory, S_IFDIR, 3,
1060157                       ps->euid, ps->egid);
1060158 if (status != 0)
1060159     {
1060160         inode_put (inode_directory);
1060161         return (NULL);
1060162     }
1060163 //
1060164 // If the inode to link was not specified, it must
1060165 // be created.
1060166 // From now on, the inode is referenced with the
1060167 // variable
1060168 // 'inode_new'.
1060169 //
1060170 inode_new = inode;
1060171 //
1060172 if (inode_new == NULL)
1060173     {
1060174         inode_new =
1060175             inode_alloc (inode_directory->sb->device, mode,
1060176                         ps->euid, ps->egid);
1060177         if (inode_new == NULL)
1060178             {
1060179                 //
1060180                 // The inode allocation failed, so, also the
1060181                 // directory
1060182                 // must be released, before return.
1060183                 //
1060184                 inode_put (inode_directory);
1060185                 return (NULL);
1060186             }
1060187     }
1060188 //
```

```
1060189 // Read the directory content and try to add the new
1060190 // item.
1060191 //
1060192 for (start = 0;
1060193      start < inode_directory->size;
1060194      start += inode_directory->sb->blksize)
1060195 {
1060196     size_read =
1060197         inode_file_read (inode_directory, start,
1060198                         buffer,
1060199                         inode_directory->sb->blksize,
1060200                         NULL);
1060201     if (size_read < sizeof (directory_t))
1060202     {
1060203         break;
1060204     }
1060205     //
1060206     // Scan the directory portion just read, for an
1060207     // unused item.
1060208     //
1060209     dir = (directory_t *) buffer;
1060210     for (d = 0; d < size_read;
1060211          d += (sizeof (directory_t)), dir++)
1060212     {
1060213         if (dir->ino == 0)
1060214         {
1060215             //
1060216             // Found an empty directory item: link
1060217             // the inode.
1060218             //
1060219             dir->ino = inode_new->ino;
1060220             strncpy (dir->name, path_name, NAME_MAX);
1060221             inode_new->links++;
1060222             inode_new->changed = 1;
1060223             //
1060224             // Update the directory inside the file
1060225             // system.
```

```
1060226         //
1060227         size_written =
1060228             inode_file_write (inode_directory,
1060229                             start, buffer, size_read);
1060230         if (size_written != size_read)
1060231             {
1060232                 //
1060233                 // Write problem: release the
1060234                 // directory and return.
1060235                 //
1060236                 inode_put (inode_directory);
1060237                 errset (EUNKNOWN);
1060238                 return (NULL);
1060239             }
1060240         //
1060241         // Save the new inode, release the
1060242         // directory and return
1060243         // the linked inode.
1060244         //
1060245         inode_save (inode_new);
1060246         inode_put (inode_directory);
1060247         return (inode_new);
1060248     }
1060249 }
1060250 }
1060251 //
1060252 // The directory don't have a free item and one must
1060253 // be appended.
1060254 //
1060255 dir = (directory_t *) buffer;
1060256 start = inode_directory->size;
1060257 //
1060258 // Prepare the buffer with the link.
1060259 //
1060260 dir->ino = inode_new->ino;
1060261 strncpy (dir->name, path_name, NAME_MAX);
1060262 inode_new->links++;
```

```

1060263     inode_new->changed = 1;
1060264     //
1060265     // Append the buffer to the directory.
1060266     //
1060267     size_written =
1060268         inode_file_write (inode_directory, start, buffer,
1060269                          (sizeof (directory_t)));
1060270     if (size_written != (sizeof (directory_t)))
1060271     {
1060272         //
1060273         // Problem updating the directory: release it
1060274         // and return.
1060275         //
1060276         inode_put (inode_directory);
1060277         errset (EUNKNOWN);
1060278         return (NULL);
1060279     }
1060280     //
1060281     // Close access to the directory inode and save the
1060282     // other inode,
1060283     // with updated link count.
1060284     //
1060285     inode_put (inode_directory);
1060286     inode_save (inode_new);
1060287     //
1060288     // Return successfully.
1060289     //
1060290     return (inode_new);
1060291 }

```

94.5.32 kernel/fs/sb_inode_status.c

Si veda la sezione [93.6.26](#).

```

1070001 #include <kernel/fs.h>
1070002 #include <errno.h>
1070003 //-----

```

```
1070004 int
1070005 sb_inode_status (sb_t * sb, ino_t ino)
1070006 {
1070007     int map_element;
1070008     int map_bit;
1070009     int map_mask;
1070010     //
1070011     // Check arguments.
1070012     //
1070013     if (ino == 0 || sb == NULL)
1070014     {
1070015         errset (EINVAL); // Invalid argument.
1070016         return (-1);
1070017     }
1070018     //
1070019     // Calculate the map element, the map bit and the
1070020     // map mask.
1070021     //
1070022     map_element = ino / 16;
1070023     map_bit = ino % 16;
1070024     map_mask = 1 << map_bit;
1070025     //
1070026     // Check the inode and return.
1070027     //
1070028     if (sb->map_inode[map_element] & map_mask)
1070029     {
1070030         return (1); // True.
1070031     }
1070032     else
1070033     {
1070034         return (0); // False.
1070035     }
1070036 }
```

94.5.33 kernel/fs/sb_mount.c



Si veda la sezione [93.6.27](#).

```
1080001 #include <kernel/fs.h>
1080002 #include <errno.h>
1080003 #include <kernel/dev.h>
1080004 #include <kernel/lib_k.h>
1080005 #include <kernel/dm.h>
1080006 #include <kernel/part.h>
1080007 //-----
1080008 sb_t *
1080009 sb_mount (dev_t device, inode_t ** inode_mnt, int options)
1080010 {
1080011     sb_t *sb;
1080012     ssize_t size_read;
1080013     addr_t start;
1080014     int m;
1080015     size_t size_sb;
1080016     size_t size_map;
1080017     int dev_major = major (device);
1080018     int dev_minor = minor (device);
1080019     int p = dev_minor & 0x000F;
1080020     int d = ((dev_minor & 0x00F0) >> 4);
1080021     //
1080022     // Find if it is already mounted.
1080023     //
1080024     sb = sb_reference (device);
1080025     if (sb != NULL)
1080026     {
1080027         errset (EBUSY);    // Device or resource busy:
1080028         // device
1080029         return (NULL);    // already mounted.
1080030     }
1080031     //
1080032     // Find if '*inode_mnt' is already mounting
1080033     // something.
1080034     //
```

```
1080035     if (*inode_mnt != NULL
1080036         && (*inode_mnt)->sb_attached != NULL)
1080037     {
1080038         errset (EBUSY);    // Device or resource busy:
1080039         // mount point
1080040         return (NULL);    // already used.
1080041     }
1080042     //
1080043     // If it is a partition, find if it can be mounted.
1080044     //
1080045     if ((dev_major == DEV_DM_MAJOR) && (p));
1080046     {
1080047         //
1080048         // It is a partition.
1080049         //
1080050         if (dm_table[d].part[p].type != PART_TYPE_MINIX)
1080051             {
1080052                 errset (E_PART_TYPE_NOT_MINIX); // Not Minix!
1080053                 return (NULL); // Cannot mount.
1080054             }
1080055     }
1080056     //
1080057     // The inode is not yet mounting anything, or it is
1080058     // new: find a free
1080059     // slot inside the super block table.
1080060     //
1080061     sb = sb_reference ((dev_t) - 1);
1080062     if (sb == NULL)
1080063     {
1080064         errset (EBUSY);    // Device or resource busy:
1080065         return (NULL);    // no free slots.
1080066     }
1080067     //
1080068     // A free slot was found: the super block header
1080069     // must be loaded, but
1080070     // before it is necessary to calculate the header
1080071     // size to be read.
```



```
1080072 //
1080073 size_sb = offsetof (sb_t, device);
1080074 //
1080075 // Then fix the starting point.
1080076 //
1080077 start = 1024; // After boot block.
1080078 //
1080079 // Read the file system super block header.
1080080 //
1080081 size_read =
1080082     dev_io ((pid_t) 0, device, DEV_READ, start, sb,
1080083             size_sb, NULL);
1080084 if (size_read != size_sb)
1080085     {
1080086         errset (EIO); // I/O error.
1080087         return (NULL);
1080088     }
1080089 //
1080090 // Save some more data.
1080091 //
1080092 sb->device = device;
1080093 sb->options = options;
1080094 sb->inode_mounted_on = *inode_mnt;
1080095 sb->blksize = (1024 << sb->log2_size_zone);
1080096 //
1080097 // Check if the super block data is valid.
1080098 //
1080099 if (sb->magic_number != 0x137F)
1080100     {
1080101         errset (ENODEV); // No such device: unsupported
1080102         sb->device = 0; // file system type.
1080103         return (NULL);
1080104     }
1080105 if (sb->map_inode_blocks > SB_MAX_INODE_BLOCKS)
1080106     {
1080107         errset (E_MAP_INODE_TOO_BIG);
1080108         return (NULL);
```

```
1080109     }
1080110     if (sb->map_zone_blocks > SB_MAX_ZONE_BLOCKS)
1080111     {
1080112         errset (E_MAP_ZONE_TOO_BIG);
1080113         return (NULL);
1080114     }
1080115     if (sb->blksize > SB_MAX_ZONE_SIZE)
1080116     {
1080117         errset (E_DATA_ZONE_TOO_BIG);
1080118         return (NULL);
1080119     }
1080120     //
1080121     // A right super block header was loaded from disk,
1080122     // now load the super block inode bit map.
1080123     //
1080124     start = 1024; // After boot block.
1080125     start += 1024; // After super block.
1080126     //
1080127     // Reset map in memory before loading.
1080128     //
1080129     for (m = 0; m < SB_MAP_INODE_SIZE; m++) // [2]
1080130     {
1080131         sb->map_inode[m] = 0xFFFF; // [2]
1080132     }
1080133     size_map = sb->map_inode_blocks * 1024;
1080134     size_read =
1080135         dev_io ((pid_t) - 1, sb->device, DEV_READ, start,
1080136             sb->map_inode, size_map, NULL);
1080137     if (size_read != size_map)
1080138     {
1080139         errset (EIO); // I/O error.
1080140         return (NULL);
1080141     }
1080142     //
1080143     // Load the super block zone bit map.
1080144     //
1080145     // After boot block:
```

```
1080146 //
1080147 start = 1024;
1080148 //
1080149 // After the super block:
1080150 //
1080151 start += 1024;
1080152 //
1080153 // After inode bit map:
1080154 //
1080155 start += (sb->map_inode_blocks * 1024);
1080156 //
1080157 // Reset map in memory, before loading.
1080158 //
1080159 for (m = 0; m < SB_MAP_ZONE_SIZE; m++)
1080160 {
1080161     sb->map_zone[m] = 0xFFFF;
1080162 }
1080163 //
1080164 size_map = sb->map_zone_blocks * 1024;
1080165 size_read =
1080166     dev_io ((pid_t) - 1, sb->device, DEV_READ, start,
1080167           sb->map_zone, size_map, NULL);
1080168 if (size_read != size_map)
1080169 {
1080170     errset (EIO); // I/O error.
1080171     return (NULL);
1080172 }
1080173 //
1080174 // Check the inode that should mount the super
1080175 // block. If '*inode_mnt' is 'NULL', then it is meant
1080176 // to be the first mount of the root file system.
1080177 // In such case, the inode must be loaded too,
1080178 // and the value for '*inode_mnt' must be modified.
1080179 //
1080180 if (*inode_mnt == NULL)
1080181 {
1080182     *inode_mnt = inode_get (device, 1);
```

```
1080183     }
1080184     //
1080185     // Check for a valid value.
1080186     //
1080187     if (*inode_mnt == NULL)
1080188     {
1080189         //
1080190         // This is bad!
1080191         //
1080192         errset (EUNKNOWN);           // Unknown error.
1080193         return (NULL);
1080194     }
1080195     //
1080196     // A valid inode is available for the mount.
1080197     //
1080198     (*inode_mnt)->sb_attached = sb;
1080199     //
1080200     // Update the super block too.
1080201     //
1080202     sb->inode_mounted_on = *inode_mnt;
1080203     //
1080204     // Return the super block pointer.
1080205     //
1080206     return (sb);
1080207 }
```

94.5.34 kernel/fs/sb_print.c



Si veda la sezione [93.6.28](#).

```
1090001 #include <sys/os32.h>
1090002 #include <kernel/fs.h>
1090003 #include <kernel/lib_k.h>
1090004 //-----
1090005 void
1090006 sb_print (void)
1090007 {
```

```

1090008     int s;
1090009     //
1090010     k_printf
1090011         ("      mnt                1st zone max file "
1090012         " inode zone  \n");
1090013     k_printf
1090014         ("dev  dev  inodes blocks dz  size size KiB "
1090015         "blocks blocks\n");
1090016     //
1090017     for (s = 0; s < SB_MAX_SLOTS; s++)
1090018     {
1090019         if (sb_table[s].device == 0)
1090020         {
1090021             continue;
1090022         }
1090023         k_printf
1090024             ("%04x %04x % 6i % 6i % 3i "
1090025             "% 4i % 8i % 6i % 6i\n",
1090026             sb_table[s].device,
1090027             sb_table[s].inode_mounted_on->sb_attached->device,
1090028             sb_table[s].inodes, sb_table[s].zones,
1090029             sb_table[s].first_data_zone,
1090030             (1024 << sb_table[s].log2_size_zone),
1090031             sb_table[s].max_file_size / 1024,
1090032             sb_table[s].map_inode_blocks,
1090033             sb_table[s].map_zone_blocks);
1090034     }
1090035 }

```

94.5.35 kernel/fs/sb_reference.c

Si veda la sezione [93.6.29](#).

```

1100001 #include <kernel/fs.h>
1100002 #include <errno.h>
1100003 //-----
1100004 sb_t *

```

```
110005 sb_reference (dev_t device)
110006 {
110007     int s;          // Slot index.
110008     //
110009     // If device is zero, a reference to the whole table
110010     // is returned.
110011     //
110012     if (device == 0)
110013     {
110014         return (sb_table);
110015     }
110016     //
110017     // If device is ((dev_t) -1), a reference to a free
110018     // slot is
110019     // returned.
110020     //
110021     if (device == ((dev_t) - 1))
110022     {
110023         for (s = 0; s < SB_MAX_SLOTS; s++)
110024         {
110025             if (sb_table[s].device == 0)
110026             {
110027                 return (&sb_table[s]);
110028             }
110029         }
110030         return (NULL);
110031     }
110032     //
110033     // A device was selected: find the super block
110034     // associated to it.
110035     //
110036     for (s = 0; s < SB_MAX_SLOTS; s++)
110037     {
110038         if (sb_table[s].device == device)
110039         {
110040             return (&sb_table[s]);
110041         }
110042     }
110043 }
```

```
1100042     }
1100043     //
1100044     // The super block was not found.
1100045     //
1100046     return (NULL);
1100047 }
```

94.5.36 kernel/fs/sb_save.c



Si veda la sezione [93.6.30](#).

```
1110001 #include <kernel/fs.h>
1110002 #include <errno.h>
1110003 #include <kernel/dev.h>
1110004 //-----
1110005 int
1110006 sb_save (sb_t * sb)
1110007 {
1110008     ssize_t size_written;
1110009     addr_t start;
1110010     size_t size_map;
1110011     //
1110012     // Check for valid argument.
1110013     //
1110014     if (sb == NULL)
1110015     {
1110016         errset (EINVAL); // Invalid argument.
1110017         return (-1);
1110018     }
1110019     //
1110020     // Check if the super block changed for some reason
1110021     // (only the inode and the zone maps can change
1110022     // really).
1110023     //
1110024     if (!sb->changed)
1110025     {
1110026         //
```

```
1110027     // Nothing to save.
1110028     //
1110029     return (0);
1110030 }
1110031 //
1110032 // Something inside the super block changed: start
1110033 // the procedure to save the inode map (recall that
1110034 // the super block header is not saved, because it
1110035 // never changes).
1110036 //
1110037 start = 1024; // After boot block.
1110038 start += 1024; // After super block.
1110039 size_map = sb->map_inode_blocks * 1024;
1110040 size_written =
1110041     dev_io ((pid_t) - 1, sb->device, DEV_WRITE, start,
1110042            sb->map_inode, size_map, NULL);
1110043 if (size_written != size_map)
1110044 {
1110045     //
1110046     // Error writing the map.
1110047     //
1110048     errset (EIO); // I/O error.
1110049     return (-1);
1110050 }
1110051 //
1110052 // Start the procedure to save the zone map.
1110053 //
1110054 start = 1024; // After boot block.
1110055 start += 1024; // After super block.
1110056 start += (sb->map_inode_blocks * 1024); // After
1110057 // inode bit
1110058 // map.
1110059 size_map = sb->map_zone_blocks * 1024;
1110060 size_written =
1110061     dev_io ((pid_t) - 1, sb->device, DEV_WRITE, start,
1110062            sb->map_zone, size_map, NULL);
1110063 if (size_written != size_map)
```



```
1110064     {
1110065         //
1110066         // Error writing the map.
1110067         //
1110068         errset (EIO);      // I/O error.
1110069         return (-1);
1110070     }
1110071     //
1110072     // Super block saved.
1110073     //
1110074     sb->changed = 0;
1110075     //
1110076     return (0);
1110077 }
```

94.5.37 kernel/fs/sb_zone_status.c

Si veda la sezione [93.6.26](#).

```
1120001 #include <kernel/fs.h>
1120002 #include <errno.h>
1120003 //-----
1120004 int
1120005 sb_zone_status (sb_t * sb, zno_t zone)
1120006 {
1120007     int map_element;
1120008     int map_bit;
1120009     int map_mask;
1120010     //
1120011     // Check arguments.
1120012     //
1120013     if (zone == 0 || sb == NULL)
1120014     {
1120015         errset (EINVAL); // Invalid argument.
1120016         return (-1);
1120017     }
1120018     //
```

```
1120019 // Calculate the map element, the map bit and the
1120020 // map mask.
1120021 //
1120022 map_element = zone / 16;
1120023 map_bit = zone % 16;
1120024 map_mask = 1 << map_bit;
1120025 //
1120026 // Check the zone and return.
1120027 //
1120028 if (sb->map_zone[map_element] & map_mask)
1120029 {
1120030     return (1); // True.
1120031 }
1120032 else
1120033 {
1120034     return (0); // False.
1120035 }
1120036 }
```

94.5.38 kernel/fs/sock_free_port.c

<<

Si veda la sezione [93.6.32](#).

```
1130001 #include <kernel/fs.h>
1130002 #include <errno.h>
1130003 #include <fcntl.h>
1130004 //-----
1130005 h_port_t
1130006 sock_free_port (void)
1130007 {
1130008     int skn;
1130009     h_port_t lport;
1130010     //
1130011     for (lport = 0xFFFF; lport >= 1024; lport--)
1130012     {
1130013         for (skn = 0; skn < SOCK_MAX_SLOTS; skn++)
1130014         {
```

```
1130015         if (sock_table[skn].lport == lport)
1130016             {
1130017                 //
1130018                 // The port is used.
1130019                 //
1130020                 break;
1130021             }
1130022     }
1130023     if (sock_table[skn].lport != lport)
1130024     {
1130025         //
1130026         // The port is new.
1130027         //
1130028         return (lport);
1130029     }
1130030 }
1130031 //
1130032 // If we are here, no free port was found.
1130033 //
1130034 return ((h_port_t) 0);
1130035 }
```

94.5.39 kernel/fs/sock_reference.c



Si veda la sezione [93.6.33](#).

```
1140001 #include <kernel/fs.h>
1140002 #include <errno.h>
1140003 #include <fcntl.h>
1140004 //-----
1140005 sock_t *
1140006 sock_reference (int skn)
1140007 {
1140008     //
1140009     // Check type of request.
1140010     //
1140011     if (skn < 0)
```

```
1140012     {
1140013         //
1140014         // Find a free slot.
1140015         //
1140016         for (skn = 0; skn < SOCK_MAX_SLOTS; skn++)
1140017             {
1140018                 if (sock_table[skn].active == 0)
1140019                     {
1140020                         return (&sock_table[skn]);
1140021                     }
1140022             }
1140023         return (NULL);
1140024     }
1140025     else if (skn > SOCK_MAX_SLOTS)
1140026         {
1140027             return (NULL);
1140028         }
1140029     else
1140030         {
1140031             return (&sock_table[skn]);
1140032         }
1140033 }
```

94.5.40 kernel/fs/zone_alloc.c



Si veda la sezione [93.6.34](#).

```
1150001 #include <kernel/fs.h>
1150002 #include <kernel/dev.h>
1150003 #include <errno.h>
1150004 //-----
1150005 zno_t
1150006 zone_alloc (sb_t * sb)
1150007 {
1150008     int m;           // Index inside the inode map.
1150009     int map_element;
1150010     int map_bit;
```

```
1150011 int map_mask;
1150012 zno_t zone;
1150013 char buffer[SB_MAX_ZONE_SIZE];
1150014 int status;
1150015 //
1150016 // Verify if write is allowed.
1150017 //
1150018 if (sb->options & MOUNT_RO)
1150019 {
1150020     errset (EROFS); // Read-only file system.
1150021     return ((zno_t) 0);
1150022 }
1150023 //
1150024 // Write allowed: scan the zone map, to find a free
1150025 // zone. If a free zone can be found, allocate it
1150026 // inside the map.
1150027 // Index 'm' starts from one, because the first bit
1150028 // of the map is reserved for a 'zero' data-zone
1150029 // that does not exist: the second bit is for the
1150030 // real first data-zone.
1150031 //
1150032 for (zone = 0, m = 1; m < (SB_MAP_ZONE_SIZE * 16); m++)
1150033 {
1150034     map_element = m / 16;
1150035     map_bit = m % 16;
1150036     map_mask = 1 << map_bit;
1150037     if (!(sb->map_zone[map_element] & map_mask))
1150038     {
1150039         //
1150040         // Found a free place: set the map.
1150041         //
1150042         sb->map_zone[map_element] |= map_mask;
1150043         sb->changed = 1;
1150044         //
1150045         // The *second* bit inside the map is for
1150046         // the first data zone (the zone after the
1150047         // inode table inside the file system),
```

```
1150048 // because the first is for a special
1150049 // 'zero' data zone, not really used.
1150050 //
1150051 zone = sb->first_data_zone + m - 1; // Found
1150052 // a free
1150053 // zone.
1150054 //
1150055 // If the zone is outside the disk size, let
1150056 // set the map bit, but reset variable
1150057 // 'zone'.
1150058 //
1150059 if (zone >= sb->zones)
1150060 {
1150061     zone = 0;
1150062 }
1150063 else
1150064 {
1150065     break;
1150066 }
1150067 }
1150068 }
1150069 if (zone == 0)
1150070 {
1150071     errset (ENOSPC); // No space left on device.
1150072     return ((zno_t) 0);
1150073 }
1150074 //
1150075 // A free zone was found and the map was modified
1150076 // inside
1150077 // the super block in memory. The zone must be
1150078 // cleared.
1150079 //
1150080 status = zone_write (sb, zone, buffer);
1150081 if (status != 0)
1150082 {
1150083     zone_free (sb, zone);
1150084     return ((zno_t) 0);
```

```
1150085     }
1150086     //
1150087     // A zone was allocated: return the number.
1150088     //
1150089     return (zone);
1150090 }
```

94.5.41 kernel/fs/zone_free.c

<<

Si veda la sezione [93.6.34](#).

```
1160001 #include <kernel/fs.h>
1160002 #include <kernel/dev.h>
1160003 #include <errno.h>
1160004 //-----
1160005 int
1160006 zone_free (sb_t * sb, zno_t zone)
1160007 {
1160008     int map_element;
1160009     int map_bit;
1160010     int map_mask;
1160011     //
1160012     // Check arguments.
1160013     //
1160014     if (sb == NULL || zone < sb->first_data_zone)
1160015     {
1160016         errset (EINVAL); // Invalid argument.
1160017         return (-1);
1160018     }
1160019     //
1160020     // Calculate the map element, the map bit and the
1160021     // map mask.
1160022     //
1160023     // The *second* bit inside the map is for the first
1160024     // data-zone
1160025     // (the zone after the inode table inside the file
1160026     // system),
```

```
1160027 // because the first is for a special 'zero'
1160028 // data-zone, not
1160029 // really used.
1160030 //
1160031 map_element = (zone - sb->first_data_zone + 1) / 16;
1160032 map_bit = (zone - sb->first_data_zone + 1) % 16;
1160033 map_mask = 1 << map_bit;
1160034 //
1160035 // Verify if the requested zone is inside the file
1160036 // system area.
1160037 //
1160038 if (zone >= sb->zones)
1160039 {
1160040     errset (EINVAL); // Invalid argument.
1160041     return (-1);
1160042 }
1160043 //
1160044 // Free the zone and return.
1160045 //
1160046 if (sb->map_zone[map_element] & map_mask)
1160047 {
1160048     sb->map_zone[map_element] &= ~map_mask;
1160049     sb->changed = 1;
1160050     return (0);
1160051 }
1160052 else
1160053 {
1160054     errset (EUNKNOWN); // The zone was
1160055     // already free.
1160056     return (-1);
1160057 }
1160058 }
```


94.5.42 kernel/fs/zone_print.c



Si veda la sezione [93.6.36](#).

```
1170001 #include <sys/os32.h>
1170002 #include <kernel/fs.h>
1170003 #include <kernel/dev.h>
1170004 #include <errno.h>
1170005 //-----
1170006 void
1170007 zone_print (sb_t * sb, zno_t zone)
1170008 {
1170009     char buffer[SB_MAX_ZONE_SIZE];
1170010     int status;
1170011     int i;
1170012     int x;
1170013     //
1170014     status = zone_read (sb, zone, buffer);
1170015     //
1170016     if (status < 0)
1170017     {
1170018         k_perror (NULL);
1170019         return;
1170020     }
1170021     //
1170022     // Print.
1170023     //
1170024     k_printf
1170025     ("dev: 0x%04x, first dzone: %i zone read: %i\n",
1170026     sb->device, sb->first_data_zone, zone);
1170027     //
1170028     // Will print at most the first 256 byte only!
1170029     //
1170030     for (i = 0; i < sb->blksize && i < 256; i++)
1170031     {
1170032         k_printf ("%02x ", buffer[i]);
1170033         x = (i + 1) % 4;
1170034         if (x == 0 && i > 0)
```

```
1170035     {
1170036         k_printf ("| ");
1170037     }
1170038     x = (i + 1) % 16;
1170039     if (x == 0 && i > 0)
1170040     {
1170041         k_printf ("\n");
1170042     }
1170043 }
1170044 }
```

94.5.43 kernel/fs/zone_read.c

<<

Si veda la sezione [93.6.37](#).

```
1180001 #include <sys/os32.h>
1180002 #include <kernel/fs.h>
1180003 #include <kernel/dev.h>
1180004 #include <errno.h>
1180005 //-----
1180006 int
1180007 zone_read (sb_t * sb, zno_t zone, void *buffer)
1180008 {
1180009     size_t size_zone;
1180010     off_t off_start;
1180011     ssize_t size_read;
1180012     //
1180013     // Verify if the requested zone is inside the file
1180014     // system area.
1180015     //
1180016     if (zone >= sb->zones)
1180017     {
1180018         errset (EINVAL); // Invalid argument.
1180019         return (-1);
1180020     }
1180021     //
1180022     // Calculate start position.
```

```

1180023 //
1180024 size_zone = 1024 << sb->log2_size_zone;
1180025 off_start = zone;
1180026 off_start *= size_zone;
1180027 //
1180028 // Read from device to the buffer.
1180029 //
1180030 size_read =
1180031     dev_io ((pid_t) - 1, sb->device, DEV_READ,
1180032           off_start, buffer, size_zone, NULL);
1180033 if (size_read != size_zone)
1180034     {
1180035         errset (EIO);      // I/O error.
1180036         return (-1);
1180037     }
1180038 else
1180039     {
1180040         return (0);
1180041     }
1180042 }

```

94.5.44 kernel/fs/zone_write.c

Si veda la sezione [93.6.37](#).

```

1190001 #include <kernel/fs.h>
1190002 #include <kernel/dev.h>
1190003 #include <errno.h>
1190004 //-----
1190005 int
1190006 zone_write (sb_t * sb, zno_t zone, void *buffer)
1190007 {
1190008     size_t size_zone;
1190009     off_t off_start;
1190010     ssize_t size_written;
1190011 //
1190012 // Verify if write is allowed.

```



```
1190013 //
1190014 if (sb->options & MOUNT_RO)
1190015 {
1190016     errset (EROFS); // Read-only file system.
1190017     return (-1);
1190018 }
1190019 //
1190020 // Verify if the requested zone is inside the file
1190021 // system area.
1190022 //
1190023 if (zone >= sb->zones)
1190024 {
1190025     errset (EINVAL); // Invalid argument.
1190026     return (-1);
1190027 }
1190028 //
1190029 // Write is allowed: calculate start position.
1190030 //
1190031 size_zone = 1024 << sb->log2_size_zone;
1190032 off_start = zone;
1190033 off_start *= size_zone;
1190034 //
1190035 // Write the buffer to the device.
1190036 //
1190037 size_written =
1190038     dev_io ((pid_t) - 1, sb->device, DEV_WRITE,
1190039             off_start, buffer, size_zone, NULL);
1190040 if (size_written != size_zone)
1190041 {
1190042     errset (EIO); // I/O error.
1190043     return (-1);
1190044 }
1190045 else
1190046 {
1190047     return (0);
1190048 }
```

1190049

}

94.6 os32: «kernel/ibm_i386.h»



Si veda la sezione [93.7](#).

```
1200001 #ifndef _KERNEL_IBM_I386_H
1200002 #define _KERNEL_IBM_I386_H 1
1200003 //-----
1200004 #include <stdint.h>
1200005 #include <inttypes.h>
1200006 #include <stdbool.h>
1200007 #include <stdarg.h>
1200008 //-----
1200009 // GDT
1200010 //-----
1200011 #define GDT_ITEMS 256 // Max is 8192 items.
1200012 //
1200013 typedef struct
1200014 {
1200015     uint32_t limit_a:16, base_a:16;
1200016     uint32_t base_b:8,
1200017         accessed:1,
1200018         write_execute:1,
1200019         expansion_conforming:1,
1200020         code_or_data:1,
1200021         code_data_or_system:1,
1200022         dpl:2,
1200023         present:1,
1200024         limit_b:4,
1200025         available:1, reserved:1, big:1, granularity:1, base_c:8;
1200026 } gdt_t;
1200027 //
1200028 extern gdt_t gdt_table[GDT_ITEMS];
1200029 //-----
1200030 typedef struct
1200031 {
```

```
120032     uint16_t limit;
120033     uint32_t base;
120034 } __attribute__((packed)) gdtr_t;      // [1]
120035 //
120036 extern gdtr_t gdt_register;
120037 //
120038 // [1] It is necessary that the structure be compact,
120039 //      so that it uses exactly 48 bits. That is why the
120040 //      attribute 'packed' for the GNU C compiler is
120041 //      used.
120042 //-----
120043 int gdt_segment (int segment, uint32_t base,
120044                 uint32_t limit, bool present,
120045                 bool code, unsigned char dpl);
120046 //
120047 void gdt_print (void *gdtr, unsigned int first,
120048               unsigned int last);
120049 void gdt_load (void *gdtr);
120050 void gdt (void);
120051 //
120052 // Segment 0 is not used,
120053 // segment 1 is for kernel code,
120054 // segment 2 is for kernel data,
120055 // segment 3 is for process 1 code,
120056 // segment 4 is for process 1 data,
120057 // ...
120058 //
120059 #define gdt_pid_to_segment_text(p) (p*2+1)
120060 #define gdt_pid_to_segment_data(p) (p*2+2)
120061 #define gdt_segment_text_to_pid(s) (s/2)
120062 #define gdt_segment_data_to_pid(s) (s/2-1)
120063 //-----
120064 // IDT
120065 //-----
120066 #define IDT_ITEMS      129      // 0-128 0x00-0x80
120067 //-----
120068 typedef struct
```

```
120069 {
120070     uint32_t offset_a:16, selector:16;
120071     uint32_t filler:8,
120072         type:4, system:1, dpl:2, present:1, offset_b:16;
120073 } idt_t;
120074 //
120075 extern idt_t idt_table[IDT_ITEMS];
120076 //-----
120077 typedef struct
120078 {
120079     uint16_t limit;
120080     uint32_t base;
120081 } __attribute__((packed)) idtr_t;
120082 //
120083 extern idtr_t idt_register;
120084 //-----
120085 void idt_descriptor (int desc, void *isr,
120086                     uint16_t selector, bool present,
120087                     char type, char dpl);
120088 void idt_load (void *idtr);
120089 void idt (void);
120090 void idt_irq_remap (unsigned int offset_1,
120091                   unsigned int offset_2);
120092 void idt_print (void *idtr, unsigned int first,
120093               unsigned int last);
120094 //-----
120095 // ISR
120096 //-----
120097 void isr_0 (void);
120098 void isr_1 (void);
120099 void isr_2 (void);
120100 void isr_3 (void);
120101 void isr_4 (void);
120102 void isr_5 (void);
120103 void isr_6 (void);
120104 void isr_7 (void);
120105 void isr_8 (void);
```

```
1200106 void isr_9 (void);
1200107 void isr_10 (void);
1200108 void isr_11 (void);
1200109 void isr_12 (void);
1200110 void isr_13 (void);
1200111 void isr_14 (void);
1200112 void isr_15 (void);
1200113 void isr_16 (void);
1200114 void isr_17 (void);
1200115 void isr_18 (void);
1200116 void isr_19 (void);
1200117 void isr_20 (void);
1200118 void isr_21 (void);
1200119 void isr_22 (void);
1200120 void isr_23 (void);
1200121 void isr_24 (void);
1200122 void isr_25 (void);
1200123 void isr_26 (void);
1200124 void isr_27 (void);
1200125 void isr_28 (void);
1200126 void isr_29 (void);
1200127 void isr_30 (void);
1200128 void isr_31 (void);
1200129 void isr_32 (void);
1200130 void isr_33 (void);
1200131 void isr_34 (void);
1200132 void isr_35 (void);
1200133 void isr_36 (void);
1200134 void isr_37 (void);
1200135 void isr_38 (void);
1200136 void isr_39 (void);
1200137 void isr_40 (void);
1200138 void isr_41 (void);
1200139 void isr_42 (void);
1200140 void isr_43 (void);
1200141 void isr_44 (void);
1200142 void isr_45 (void);
```



```
1200143 void isr_46 (void);
1200144 void isr_47 (void);
1200145 void isr_128 (void);
1200146 //
1200147 char *isr_exception_name (int exception);
1200148 //
1200149 void isr_exception_unrecoverable (uint32_t eax,
1200150                                   uint32_t ecx,
1200151                                   uint32_t edx,
1200152                                   uint32_t ebx,
1200153                                   uint32_t ebp,
1200154                                   uint32_t esi,
1200155                                   uint32_t edi,
1200156                                   uint32_t ds,
1200157                                   uint32_t es,
1200158                                   uint32_t fs,
1200159                                   uint32_t gs,
1200160                                   uint32_t interrupt,
1200161                                   uint32_t error,
1200162                                   uint32_t eip,
1200163                                   uint32_t cs,
1200164                                   uint32_t eflags);
1200165 //
1200166 void isr_irq_clear (uint32_t idtn);
1200167 void isr_irq_clear_pic1 (void);
1200168 void isr_irq_clear_pic2 (void);
1200169 //-----
1200170 // I/O
1200171 //-----
1200172 uint32_t _in_8 (uint32_t port);
1200173 uint32_t _in_16 (uint32_t port);
1200174 uint32_t _in_32 (uint32_t port);
1200175 void _out_8 (uint32_t port, uint32_t value);
1200176 void _out_16 (uint32_t port, uint32_t value);
1200177 void _out_32 (uint32_t port, uint32_t value);
1200178 //
1200179 #define in_8(port) \
```

```

1200180      ((unsigned int) _in_8 ((uint32_t) port))
1200181 #define in_16(port) \
1200182      ((unsigned int) _in_16 ((uint32_t) port))
1200183 #define in_32(port) \
1200184      ((unsigned int) _in_32 ((uint32_t) port))
1200185 #define out_8(port, value) \
1200186      (_out_8 ((uint32_t) port, (uint32_t) value))
1200187 #define out_16(port, value) \
1200188      (_out_16 ((uint32_t) port, (uint32_t) value))
1200189 #define out_32(port, value) \
1200190      (_out_32 ((uint32_t) port, (uint32_t) value))
1200191 //-----
1200192 // Interrupt on/off
1200193 //-----
1200194 void cli (void);
1200195 void sti (void);
1200196 void irq_on (unsigned int irq);
1200197 void irq_off (unsigned int irq);
1200198 //-----
1200199 #endif

```

94.6.1	kernel/ibm_i386/_in_16.s	1283
94.6.2	kernel/ibm_i386/_in_32.s	1284
94.6.3	kernel/ibm_i386/_in_8.s	1285
94.6.4	kernel/ibm_i386/_out_16.s	1286
94.6.5	kernel/ibm_i386/_out_32.s	1287
94.6.6	kernel/ibm_i386/_out_8.s	1288
94.6.7	kernel/ibm_i386/cli.s	1288
94.6.8	kernel/ibm_i386/gdt.c	1289
94.6.9	kernel/ibm_i386/gdt_load.s	1290

- 94.6.10 kernel/ibm_i386/gdt_print.c 1291
- 94.6.11 kernel/ibm_i386/gdt_public.c 1292
- 94.6.12 kernel/ibm_i386/gdt_segment.c 1292
- 94.6.13 kernel/ibm_i386/idt.c 1294
- 94.6.14 kernel/ibm_i386/idt_descriptor.c 1296
- 94.6.15 kernel/ibm_i386/idt_irq_remap.c 1298
- 94.6.16 kernel/ibm_i386/idt_load.s 1300
- 94.6.17 kernel/ibm_i386/idt_print.c 1300
- 94.6.18 kernel/ibm_i386/idt_public.c 1301
- 94.6.19 kernel/ibm_i386/irq_off.c 1301
- 94.6.20 kernel/ibm_i386/irq_on.c 1302
- 94.6.21 kernel/ibm_i386/isr.s 1303
- 94.6.22 kernel/ibm_i386/isr_exception_name.c 1327
- 94.6.23 kernel/ibm_i386/isr_exception_unrecoverable.c .. 1328
- 94.6.24 kernel/ibm_i386/isr_irq_clear.c 1329
- 94.6.25 kernel/ibm_i386/isr_irq_clear_pic1.c 1330
- 94.6.26 kernel/ibm_i386/isr_irq_clear_pic2.c 1331
- 94.6.27 kernel/ibm_i386/sti.s 1331

94.6.1 kernel/ibm_i386/_in_16.s

Si veda la sezione [93.7](#).

1210001	.global _in_16
1210002	#-----



```

1210003  .section .text
1210004  #-----
1210005  # Port input word.
1210006  #-----
1210007  _in_16:
1210008      enter $4, $0                # 1 local variable.
1210009      pushf
1210010      pusha
1210011      .equ IN_PORT, 8              # First argument.
1210012      .equ IN_DATA, -4            # Local variable.
1210013      mov  IN_PORT(%ebp), %edx    # Copy the port number
1210014                                  # into EDX, but
1210015                                  # then only DX will be
1210016                                  # used.
1210017      mov  $0, %eax              # Reset EAX.
1210018      in   %dx, %ax              # Read DX port and
1210019                                  # save into AX.
1210020      mov  %eax, IN_DATA(%ebp)    # Save EAX inside the
1210021                                  # local variable.
1210022      popa
1210023      popf
1210024      mov  IN_DATA(%ebp), %eax    # Restore EAX and
1210025      leave                          # return.
1210026      ret

```

94.6.2 kernel/ibm_i386/_in_32.s



Si veda la sezione [93.7](#).

```

1220001  .global _in_32
1220002  #-----
1220003  .section .text
1220004  #-----
1220005  # Port input word.
1220006  #-----
1220007  _in_32:
1220008      enter $4, $0                # 1 local variable.

```

```

1220009     pushf
1220010     pusha
1220011     .equ IN_PORT, 8           # First argument.
1220012     .equ IN_DATA, -4        # Local variable.
1220013     mov  IN_PORT(%ebp), %edx # Copy the port number
1220014                               # into EDX, but
1220015                               # then only DX will be
1220016                               # used.
1220017     mov  $0, %eax           # Reset EAX.
1220018     inl  %dx, %eax         # Read DX port and
1220019                               # save into EAX.
1220020     mov  %eax, IN_DATA(%ebp) # Save EAX inside the
1220021                               # local variable.
1220022     popa
1220023     popf
1220024     mov  IN_DATA(%ebp), %eax # Restore EAX and
1220025     leave                               # return.
1220026     ret

```

94.6.3 kernel/ibm_i386/_in_8.s

Si veda la sezione [93.7](#).



```

1230001     .global _in_8
1230002     #-----
1230003     .section .text
1230004     #-----
1230005     # Port input byte.
1230006     #-----
1230007     _in_8:
1230008     enter $4, $0           # 1 local variable.
1230009     pushf
1230010     pusha
1230011     .equ IN_PORT, 8           # First argument.
1230012     .equ IN_DATA, -4        # Local variable.
1230013     mov  IN_PORT(%ebp), %edx # Copy the port number
1230014                               # into EDX, but

```

```

1230015                                     # then only DX will be
1230016                                     # used.
1230017     mov     $0, %eax                       # Reset EAX.
1230018     inb    %dx, %al                       # Read DX port and
1230019                                     # save into AL.
1230020     mov    %eax, IN_DATA(%ebp)            # Save EAX inside the
1230021                                     # local variable.
1230022     popa
1230023     popf
1230024     mov    IN_DATA(%ebp), %eax           # Restore EAX and
1230025     leave
1230026     ret

```

94.6.4 kernel/ibm_i386/_out_16.s

«

Si veda la sezione [93.7](#).

```

1240001     .global _out_16
1240002     #-----
1240003     .section .text
1240004     #-----
1240005     # Port output word.
1240006     #-----
1240007     _out_16:
1240008         enter $0, $0                       # No local variables.
1240009         pushf
1240010         pusha
1240011         .equ  OUT_PORT,  8                 # First parameter.
1240012         .equ  OUT_DATA, 12                 # Second parameter.
1240013         mov   OUT_PORT(%ebp), %edx         # Copy output port to
1240014                                     # EDX, but only DX
1240015                                     # will be used.
1240016         mov   OUT_DATA(%ebp), %eax        # Copy output data to
1240017                                     # EAX, but only AX
1240018                                     # will be used.
1240019         out   %ax, %dx                     # Send to the port.
1240020         popa

```

1240021	popf
1240022	leave
1240023	ret

94.6.5 kernel/ibm_i386/_out_32.s



Si veda la sezione [93.7](#).

1250001	.global _out_32	
1250002	#-----	
1250003	.section .text	
1250004	#-----	
1250005	# Port output word.	
1250006	#-----	
1250007	_out_32:	
1250008	enter \$0, \$0	# No local variables.
1250009	pushf	
1250010	pusha	
1250011	.equ OUT_PORT, 8	# First parameter.
1250012	.equ OUT_DATA, 12	# Second parameter.
1250013	mov OUT_PORT(%ebp), %edx	# Copy output port to
1250014		# EDX, but only DX
1250015		# will be used.
1250016	mov OUT_DATA(%ebp), %eax	# Copy output data to
1250017		# EAX.
1250018	outl %eax, %dx	# Send to the port.
1250019	popa	
1250020	popf	
1250021	leave	
1250022	ret	

94.6.6 kernel/ibm_i386/_out_8.s

<<

Si veda la sezione [93.7](#).

```
1260001  .global _out_8
1260002  #-----
1260003  .section .text
1260004  #-----
1260005  # Port output byte.
1260006  #-----
1260007  _out_8:
1260008      enter $0, $0          # No local variables.
1260009      pushf
1260010      pusha
1260011      .equ  OUT_PORT,  8      # First parameter.
1260012      .equ  OUT_DATA, 12     # Second parameter.
1260013      mov   OUT_PORT(%ebp), %edx # Copy output port to
1260014                                     # EDX, but only DX
1260015                                     # will be used.
1260016      mov   OUT_DATA(%ebp), %eax # Copy output data to
1260017                                     # EAX, but only AL
1260018                                     # will be used.
1260019      outb %al, %dx         # Send to the port.
1260020      popa
1260021      popf
1260022      leave
1260023      ret
```

94.6.7 kernel/ibm_i386/cli.s

<<

Si veda la sezione [93.7](#).

```
1270001  .global cli
1270002  #-----
1270003  .text
1270004  #-----
1270005  # Clear interrupt flag.
1270006  #-----
```



```
1270007 .align 4
1270008 cli:
1270009     cli
1270010     ret
```

94.6.8 kernel/ibm_i386/gdt.c

Si veda la sezione [93.7](#).

```
1280001 #include <kernel/ibm_i386.h>
1280002 #include <kernel/multiboot.h>
1280003 //-----
1280004 void
1280005 gdt (void)
1280006 {
1280007     uint32_t blocks;          // Total available memory
1280008     // blocks.
1280009     //
1280010     // Calculate memory blocks.
1280011     //
1280012     blocks = (multiboot.mem_upper * 1024) / 4096;
1280013     //
1280014     // Set data for GDTR register.
1280015     //
1280016     gdt_register.limit = (sizeof (gdt_table) - 1);
1280017     gdt_register.base = (uint32_t) & gdt_table[0];
1280018     //
1280019     // Reset items inside 'gdt_table[]'.
1280020     // gdt_table[0] must be null and is not to be
1280021     // used.
1280022     //
1280023     int i;
1280024     for (i = 0; i < GDT_ITEMS; i++)
1280025     {
1280026         gdt_segment (i, 0, 0, 0, 0, 0);
1280027     }
1280028     //
```

```

1280029 // gdt_table[1] is for kernel code.
1280030 // It covers all the available memory, with DPL 0.
1280031 // The selector is 8+0 = 0x08+0.
1280032 //
1280033 gdt_segment (1, 0, blocks, 1, 1, 0);
1280034 //
1280035 // gdt_table[2] is for kernel data, including stack
1280036 // (the stack
1280037 // MUST be in the same address space of data).
1280038 // It covers all the available memory, with DPL 0.
1280039 // The selector is 16+0 = 0x10+0.
1280040 //
1280041 gdt_segment (2, 0, blocks, 1, 0, 0);
1280042 //
1280043 // Load the GDT table.
1280044 //
1280045 gdt_load (&gdt_register);
1280046 }

```

94.6.9 kernel/ibm_i386/gdt_load.s

«

Si veda la sezione [93.7](#).

```

1290001 .globl gdt_load
1290002 #
1290003 gdt_load:
1290004     enter $0, $0
1290005     .equ gdtr_pointer, 8           # Primo argomento.
1290006     mov  gdtr_pointer(%ebp), %eax # Copia il
1290007                                     # puntatore in EAX.
1290008     leave
1290009     #
1290010     lgdt (%eax)                  # Carica il registro GDTR
1290011                                     # dall'indirizzo in EAX.
1290012     #
1290013     # 2 kernel data, included stack, DPL 0, covering
1290014     # all the available

```

```
1290015     #   memory: segment selector 0x10+0.
1290016     #
1290017     mov   $16, %ax
1290018     mov   %ax, %ds
1290019     mov   %ax, %es
1290020     mov   %ax, %fs
1290021     mov   %ax, %gs
1290022     mov   %ax, %ss           # The stack MUST be in the same
1290023                           # address space of the other
1290024                           # data, to allow pointers to
1290025     #                           # work correctly.
1290026     #
1290027     # 2 kernel code, DPL 0, covering all the available
1290028     # memory:
1290029     #   segment selector 0x08+0.
1290030     #
1290031     jmp   $8, $flush
1290032 flush:
1290033     ret
```

94.6.10 kernel/ibm_i386/gdt_print.c

Si veda la sezione [93.7](#).

```
1300001 #include <kernel/ibm_i386.h>
1300002 #include <kernel/lib_k.h>
1300003 //
1300004 void
1300005 gdt_print (void *gdtr, unsigned int first,
1300006           unsigned int last)
1300007 {
1300008     gdtr_t *g = gdtr;
1300009     uint32_t *p = (uint32_t *) g->base;
1300010     //
1300011     int max = (g->limit + 1) / (sizeof (uint32_t));
1300012     int i;
1300013     //
```



```

1300014     if (((first * 2) > max) || (first > last))
1300015     {
1300016         return;
1300017     }
1300018     //
1300019     k_printf ("%s] base: 0x%08" PRIx32 " limit: 0x%04"
1300020              "          " PRIx32 "\n", __func__, g->base, g->limit);
1300021     //
1300022     for (i = (first * 2); i < max && i <= (last * 2); i += 2)
1300023     {
1300024         k_printf ("%4" PRIx32 "] %032" PRIb32 " %032"
1300025                  "          " PRIb32 "\n", i / 2, p[i], p[i + 1]);
1300026     }
1300027 }

```

94.6.11 kernel/ibm_i386/gdt_public.c

«

Si veda la sezione [93.7](#).

```

1310001 #include <kernel/ibm_i386.h>
1310002 //-----
1310003 gdt_t gdt_table[GDT_ITEMS];
1310004 gdtr_t gdt_register;

```

94.6.12 kernel/ibm_i386/gdt_segment.c

«

Si veda la sezione [93.7](#).

```

1320001 #include <kernel/ibm_i386.h>
1320002 #include <errno.h>
1320003 //-----
1320004 int
1320005 gdt_segment (int segment,
1320006             uint32_t base,
1320007             uint32_t limit,
1320008             bool present, bool code, unsigned char dpl)
1320009 {

```

```
1320010 //
1320011 // Verify if the segment is valid.
1320012 //
1320013 if ((segment >= ((sizeof (gdt_table)) / 8))
1320014     || (segment < 0))
1320015     {
1320016         errset (EINVAL);
1320017         return (-1);
1320018     }
1320019 //
1320020 // Limit.
1320021 //
1320022 gdt_table[segment].limit_a = (limit & 0x0000FFFF);
1320023 gdt_table[segment].limit_b = limit / 0x10000;
1320024 //
1320025 // Base address.
1320026 //
1320027 gdt_table[segment].base_a = (base & 0x0000FFFF);
1320028 gdt_table[segment].base_b =
1320029     ((base / 0x10000) & 0x000000FF);
1320030 gdt_table[segment].base_c = (base / 0x1000000);
1320031 //
1320032 // Attributes.
1320033 //
1320034
1320035 //
1320036 // Internal update.
1320037 //
1320038 gdt_table[segment].accessed = 0;
1320039 //
1320040 // r, w, x.
1320041 //
1320042 gdt_table[segment].write_execute = 1;
1320043 //
1320044 // Normal and conforming.
1320045 //
1320046 gdt_table[segment].expansion_conforming = 0;
```

```

1320047 //
1320048 gdt_table[segment].code_or_data = code;
1320049 gdt_table[segment].code_data_or_system = 1;
1320050 gdt_table[segment].dpl = dpl;
1320051 gdt_table[segment].present = present;
1320052 gdt_table[segment].available = 0; // 0
1320053 gdt_table[segment].reserved = 0; // 0
1320054 gdt_table[segment].big = 1; // 32 bit
1320055 gdt_table[segment].granularity = 1; // 4 Kibyte
1320056 //
1320057 return (0);
1320058 }

```

94.6.13 kernel/ibm_i386/idt.c

«

Si veda la sezione [93.7](#).

```

1330001 #include <kernel/ibm_i386.h>
1330002 //-----
1330003 void
1330004 idt (void)
1330005 {
1330006 //
1330007 // Set necessary data for the IDTR register.
1330008 //
1330009 idt_register.limit = (sizeof (idt_table) - 1);
1330010 idt_register.base = (uint32_t) & idt_table[0];
1330011 //
1330012 // Reset all items inside the array 'idt_table[]'.
1330013 //
1330014 int i;
1330015 for (i = 0; i < IDT_ITEMS; i++)
1330016 {
1330017     idt_descriptor (i, 0, 0, 0, 0, 0);
1330018 }
1330019 //
1330020 // Place hardware interrupt from IRQ 0 to IRQ 7

```

```
1330021 // starting from descriptor 32 and from IRQ 8 to
1330022 // IRQ 15 starting from descriptor 40.
1330023 //
1330024 idt_irq_remap (32, 40);
1330025 //
1330026 // Set the ISR routines to the items inside the IDT
1330027 // table.
1330028 //
1330029 idt_descriptor (0, isr_0, 0x0008, 1, 0xE, 0);
1330030 idt_descriptor (1, isr_1, 0x0008, 1, 0xE, 0);
1330031 idt_descriptor (2, isr_2, 0x0008, 1, 0xE, 0);
1330032 idt_descriptor (3, isr_3, 0x0008, 1, 0xE, 0);
1330033 idt_descriptor (4, isr_4, 0x0008, 1, 0xE, 0);
1330034 idt_descriptor (5, isr_5, 0x0008, 1, 0xE, 0);
1330035 idt_descriptor (6, isr_6, 0x0008, 1, 0xE, 0);
1330036 idt_descriptor (7, isr_7, 0x0008, 1, 0xE, 0);
1330037 idt_descriptor (8, isr_8, 0x0008, 1, 0xE, 0);
1330038 idt_descriptor (9, isr_9, 0x0008, 1, 0xE, 0);
1330039 idt_descriptor (10, isr_10, 0x0008, 1, 0xE, 0);
1330040 idt_descriptor (11, isr_11, 0x0008, 1, 0xE, 0);
1330041 idt_descriptor (12, isr_12, 0x0008, 1, 0xE, 0);
1330042 idt_descriptor (13, isr_13, 0x0008, 1, 0xE, 0);
1330043 idt_descriptor (14, isr_14, 0x0008, 1, 0xE, 0);
1330044 idt_descriptor (15, isr_15, 0x0008, 1, 0xE, 0);
1330045 idt_descriptor (16, isr_16, 0x0008, 1, 0xE, 0);
1330046 idt_descriptor (17, isr_17, 0x0008, 1, 0xE, 0);
1330047 idt_descriptor (18, isr_18, 0x0008, 1, 0xE, 0);
1330048 idt_descriptor (19, isr_19, 0x0008, 1, 0xE, 0);
1330049 idt_descriptor (20, isr_20, 0x0008, 1, 0xE, 0);
1330050 idt_descriptor (21, isr_21, 0x0008, 1, 0xE, 0);
1330051 idt_descriptor (22, isr_22, 0x0008, 1, 0xE, 0);
1330052 idt_descriptor (23, isr_23, 0x0008, 1, 0xE, 0);
1330053 idt_descriptor (24, isr_24, 0x0008, 1, 0xE, 0);
1330054 idt_descriptor (25, isr_25, 0x0008, 1, 0xE, 0);
1330055 idt_descriptor (26, isr_26, 0x0008, 1, 0xE, 0);
1330056 idt_descriptor (27, isr_27, 0x0008, 1, 0xE, 0);
1330057 idt_descriptor (28, isr_28, 0x0008, 1, 0xE, 0);
```

```
1330058     idt_descriptor (29, isr_29, 0x0008, 1, 0xE, 0);
1330059     idt_descriptor (30, isr_10, 0x0008, 1, 0xE, 0);
1330060     idt_descriptor (31, isr_31, 0x0008, 1, 0xE, 0);
1330061     idt_descriptor (32, isr_32, 0x0008, 1, 0xE, 0);
1330062     idt_descriptor (33, isr_33, 0x0008, 1, 0xE, 0);
1330063     idt_descriptor (34, isr_34, 0x0008, 1, 0xE, 0);
1330064     idt_descriptor (35, isr_35, 0x0008, 1, 0xE, 0);
1330065     idt_descriptor (36, isr_36, 0x0008, 1, 0xE, 0);
1330066     idt_descriptor (37, isr_37, 0x0008, 1, 0xE, 0);
1330067     idt_descriptor (38, isr_38, 0x0008, 1, 0xE, 0);
1330068     idt_descriptor (39, isr_39, 0x0008, 1, 0xE, 0);
1330069     idt_descriptor (40, isr_40, 0x0008, 1, 0xE, 0);
1330070     idt_descriptor (41, isr_41, 0x0008, 1, 0xE, 0);
1330071     idt_descriptor (42, isr_42, 0x0008, 1, 0xE, 0);
1330072     idt_descriptor (43, isr_43, 0x0008, 1, 0xE, 0);
1330073     idt_descriptor (44, isr_44, 0x0008, 1, 0xE, 0);
1330074     idt_descriptor (45, isr_45, 0x0008, 1, 0xE, 0);
1330075     idt_descriptor (46, isr_46, 0x0008, 1, 0xE, 0);
1330076     idt_descriptor (47, isr_47, 0x0008, 1, 0xE, 0);
1330077     //
1330078     // The following item is for the system calls.
1330079     //
1330080     idt_descriptor (128, isr_128, 0x0008, 1, 0xE, 0);
1330081     //
1330082     // Activate the IDT loading the IDTR register.
1330083     //
1330084     idt_load (&idt_register);
1330085 }
```

94.6.14 kernel/ibm_i386/idt_descriptor.c



Si veda la sezione [93.7](#).

```
1340001 #include <kernel/ibm_i386.h>
1340002 //-----
1340003 void
1340004 idt_descriptor (int desc,
```



```
1340005         void *isr,
1340006         uint16_t selector,
1340007         bool present, char type, char dpl)
1340008     {
1340009         uint32_t offset = (uint32_t) isr;
1340010         //
1340011         // Unset reserved bits and the system bit.
1340012         //
1340013         idt_table[desc].filler = 0;
1340014         idt_table[desc].system = 0;
1340015         //
1340016         // Relative address.
1340017         //
1340018         idt_table[desc].offset_a = (offset & 0x0000FFFF);
1340019         idt_table[desc].offset_b = (offset / 0x10000);
1340020         //
1340021         // Selector.
1340022         //
1340023         idt_table[desc].selector = selector;
1340024         //
1340025         // Valid item?
1340026         //
1340027         idt_table[desc].present = present;
1340028         //
1340029         // Type (gate type).
1340030         //
1340031         idt_table[desc].type = (type & 0x0F);
1340032         //
1340033         // DPL.
1340034         //
1340035         idt_table[desc].dpl = (dpl & 0x03);
1340036     }
```

94.6.15 kernel/ibm_i386/idt_irq_remap.c

<<

Si veda la sezione 93.7.

```
1350001 #include <kernel/lib_k.h>
1350002 #include <kernel/ibm_i386.h>
1350003 //-----
1350004 #define DEBUG 0
1350005 //-----
1350006 void
1350007 idt_irq_remap (unsigned int offset_1, unsigned int offset_2)
1350008 {
1350009     //
1350010     // PIC_P è il PIC primario o «master»;
1350011     // PIC_S è il PIC secondario o «slave».
1350012     //
1350013     // Quando si manifesta un IRQ che riguarda il PIC
1350014     // secondario,
1350015     // il PIC primario riceve IRQ 2.
1350016     //
1350017     // ICW = initialization command word.
1350018     // OCW = operation command word.
1350019     //
1350020     if (DEBUG)
1350021     {
1350022         k_printf
1350023             ("[%s] PIC (programmable interrupt "
1350024              "controller) " "remap: ", __func__);
1350025     }
1350026     //
1350027     out_8 (0x20, 0x10 + 0x01);    // Initialization:
1350028     // 0x10 means that is
1350029     out_8 (0xA0, 0x10 + 0x01);    // is ICW1; 0x01 means
1350030     // that must
1350031     // continue up to ICW4.
1350032     if (DEBUG)
1350033     {
1350034         k_printf ("ICW1");
```

```
1350035     }
1350036     out_8 (0x21, offset_1);           // ICW2: PIC_P
1350037     // starting at
1350038     // «offset_1».
1350039     out_8 (0xA1, offset_2);         // PIC_S starting at
1350040     // «offset_2».
1350041     if (DEBUG)
1350042     {
1350043         k_printf (", ICW2");
1350044     }
1350045     out_8 (0x21, 0x04);           // ICW3 PIC_P: IRQ2 driven
1350046     // from PIC_S.
1350047     out_8 (0xA1, 0x02);         // ICW3 PIC_S: driving IRQ2
1350048     // from PIC_P.
1350049     if (DEBUG)
1350050     {
1350051         k_printf (", ICW3");
1350052     }
1350053     out_8 (0x21, 0x01);           // ICW4: si precisa solo la
1350054     // modalità
1350055     out_8 (0xA1, 0x01);         // del microprocessore; 0x01 =
1350056     // 8086.
1350057     if (DEBUG)
1350058     {
1350059         k_printf (", ICW4");
1350060     }
1350061     out_8 (0x21, 0x00);           // OCW1: reset mask to enable
1350062     // all
1350063     out_8 (0xA1, 0x00);         // IRQ numbers.
1350064     if (DEBUG)
1350065     {
1350066         k_printf (", OCW1.\n");
1350067     }
1350068 }
```

94.6.16 kernel/ibm_i386/idt_load.s



Si veda la sezione [93.7](#).

```
1360001  .globl  idt_load
1360002  #
1360003  idt_load:
1360004      enter $0, $0
1360005      .equ idtr_pointer, 8          # Primo argomento.
1360006      mov  idtr_pointer(%ebp), %eax # Copia il puntatore
1360007                                      # in EAX.
1360008
1360009      leave
1360010      #
1360011      lidt (%eax)                # Utilizza la tabella IDT a cui
1360012                                      # punta EAX.
1360013      #
1360014      ret
```

94.6.17 kernel/ibm_i386/idt_print.c



Si veda la sezione [93.7](#).

```
1370001  #include <kernel/ibm_i386.h>
1370002  #include <kernel/lib_k.h>
1370003  //
1370004  void
1370005  idt_print (void *idtr, unsigned int first,
1370006            unsigned int last)
1370007  {
1370008      idtr_t *g = idtr;
1370009      uint32_t *p = (uint32_t *) g->base;
1370010      //
1370011      int max = (g->limit + 1) / (sizeof (uint32_t));
1370012      int i;
1370013      //
1370014      if (((first * 2) > max) || (first > last))
1370015          {
1370016              return;
```

```

1370017     }
1370018     //
1370019     k_printf ("%s] base: 0x%08" PRIx32 " limit: 0x%04"
1370020             PRIx32 "\n", __func__, g->base, g->limit);
1370021     //
1370022     for (i = (first * 2); i < max && i <= (last * 2); i += 2)
1370023     {
1370024         k_printf ("%4" PRIx32 "]" %032" PRIB32 " %032"
1370025                 PRIB32 "\n", i / 2, p[i], p[i + 1]);
1370026     }
1370027 }

```

94.6.18 kernel/ibm_i386/idt_public.c

«

Si veda la sezione [93.7](#).

```

1380001 #include <kernel/ibm_i386.h>
1380002 //-----
1380003 idt_t idt_table[129];
1380004 idtr_t idt_register;

```

94.6.19 kernel/ibm_i386/irq_off.c

«

Si veda la sezione [93.7](#).

```

1390001 #include <kernel/ibm_i386.h>
1390002 #include <stdint.h>
1390003 //-----
1390004 void
1390005 irq_off (unsigned int irq)
1390006 {
1390007     uint32_t mask;
1390008     uint32_t status;
1390009     //
1390010     if (irq > 15)
1390011     {
1390012         return;    // There is not such IRQ.

```

```

1390013     }
1390014     else
1390015     {
1390016         mask = ((uint32_t) 1 << irq);
1390017         //
1390018         // IRQ from 0 to 7.
1390019         //
1390020         status = in_8 ((uint32_t) 0x21);
1390021         status = status | mask;
1390022         out_8 ((uint32_t) 0x21, status);
1390023         //
1390024         // IRQ from 8 to 15.
1390025         //
1390026         status = in_8 ((uint32_t) 0xA1);
1390027         status = status | (mask >> 8);
1390028         out_8 ((uint32_t) 0xA1, status);
1390029     }
1390030 }

```

94.6.20 kernel/ibm_i386/irq_on.c



Si veda la sezione [93.7](#).

```

1400001 #include <kernel/ibm_i386.h>
1400002 #include <stdint.h>
1400003 //-----
1400004 void
1400005 irq_on (unsigned int irq)
1400006 {
1400007     uint32_t mask;
1400008     uint32_t status;
1400009     //
1400010     if (irq > 15)
1400011     {
1400012         return;    // There is not such IRQ.
1400013     }
1400014     else

```

```
1400015     {
1400016         mask = ~((uint32_t) 1 << irq);
1400017         //
1400018         // IRQ from 0 to 7.
1400019         //
1400020         status = in_8 ((uint32_t) 0x21);
1400021         status = status & mask;
1400022         out_8 ((uint32_t) 0x21, status);
1400023         //
1400024         // IRQ from 8 to 15.
1400025         //
1400026         status = in_8 ((uint32_t) 0xA1);
1400027         status = status & (mask >> 8);
1400028         out_8 ((uint32_t) 0xA1, status);
1400029     }
1400030 }
```

94.6.21 kernel/ibm_i386/isr.s

Si veda la sezione [93.7](#).

```
1410001 .extern isr_exception_unrecoverable
1410002 .extern isr_irq_clear
1410003 .extern isr_irq_clear_pic1
1410004 .extern isr_irq_clear_pic2
1410005 .extern kbd_isr
1410006 .extern sysroutine
1410007 .extern proc_current
1410008 .extern proc_stack_segment_selector;
1410009 .extern proc_stack_pointer
1410010 .extern proc_scheduler
1410011 #.extern proc_sch_terminals
1410012 #
1410013 .global _clock_kernel
1410014 .global _clock_time
1410015 .global _ksp
1410016 #
```

1410017	.global isr_0
1410018	.global isr_1
1410019	.global isr_2
1410020	.global isr_3
1410021	.global isr_4
1410022	.global isr_5
1410023	.global isr_6
1410024	.global isr_7
1410025	.global isr_8
1410026	.global isr_9
1410027	.global isr_10
1410028	.global isr_11
1410029	.global isr_12
1410030	.global isr_13
1410031	.global isr_14
1410032	.global isr_15
1410033	.global isr_16
1410034	.global isr_17
1410035	.global isr_18
1410036	.global isr_19
1410037	.global isr_20
1410038	.global isr_21
1410039	.global isr_22
1410040	.global isr_23
1410041	.global isr_24
1410042	.global isr_25
1410043	.global isr_26
1410044	.global isr_27
1410045	.global isr_28
1410046	.global isr_29
1410047	.global isr_30
1410048	.global isr_31
1410049	.global isr_32
1410050	.global isr_33
1410051	.global isr_34
1410052	.global isr_35
1410053	.global isr_36


```
1410054 .global isr_37
1410055 .global isr_38
1410056 .global isr_39
1410057 .global isr_40
1410058 .global isr_41
1410059 .global isr_42
1410060 .global isr_43
1410061 .global isr_44
1410062 .global isr_45
1410063 .global isr_46
1410064 .global isr_47
1410065 .global isr_128
1410066 #-----
1410067 .section .data
1410068 #-----
1410069 proc_syscallnr:          .int    0x00000000
1410070 proc_msg_offset:        .int    0x00000000
1410071 proc_msg_size:          .int    0x00000000
1410072 proc_instruction_pointer: .int    0x00000000
1410073 proc_back_address:      .int    0x00000000
1410074 _ksp:                   .int    0x00000000
1410075 syscall_working:        .int    0x00000000
1410076 _clock_kernel:         .int    0x00000000
1410077 kticks_lo:              .int    0x00000000
1410078 kticks_hi:              .int    0x00000000
1410079 _clock_time:           .int    0x00000000
1410080 tticks_lo:              .int    0x00000000
1410081 tticks_hi:              .int    0x00000000
1410082 #-----
1410083 .section .text
1410084 #-----
1410085 #
1410086 # Inside the stack there is already, placed by the CPU:
1410087 # [omissis]
1410088 #   push %eflags
1410089 #   push %cs
1410090 #   push %eip
```

```
1410091 #
1410092 #-----
1410093 isr_0:          # «division by zero exception»
1410094     cli
1410095     push $0      # Null error code.
1410096     push $0      # Exception number.
1410097     jmp exception_unrecoverable
1410098 #-----
1410099 isr_1:          # «debug exception»
1410100     cli
1410101     push $0      # Null error code.
1410102     push $1      # Exception number.
1410103     jmp exception_unrecoverable
1410104 #-----
1410105 isr_2:          # «non maskable interrupt exception»
1410106     cli
1410107     push $0      # Null error code.
1410108     push $2      # Exception number.
1410109     jmp exception_unrecoverable
1410110 #-----
1410111 isr_3:          # «breakpoint exception»
1410112     cli
1410113     push $0      # Null error code.
1410114     push $3      # Exception number.
1410115     jmp exception_unrecoverable
1410116 #-----
1410117 isr_4:          # «into detected overflow exception»
1410118     cli
1410119     push $0      # Null error code.
1410120     push $4      # Exception number.
1410121     jmp exception_unrecoverable
1410122 #-----
1410123 isr_5:          # «out of bounds exception»
1410124     cli
1410125     push $0      # Null error code.
1410126     push $5      # Exception number.
1410127     jmp exception_unrecoverable
```

```
1410128 #-----
1410129 isr_6:          # «invalid opcode exception»
1410130     cli
1410131     push $0      # Null error code.
1410132     push $6      # Exception number.
1410133     jmp exception_unrecoverable
1410134 #-----
1410135 isr_7:          # «no coprocessor exception»
1410136     cli
1410137     push $0      # Null error code.
1410138     push $7      # Exception number.
1410139     jmp exception_unrecoverable
1410140 #-----
1410141 isr_8:          # «double fault exception»
1410142     cli
1410143     #           # Error code already present.
1410144     push $8      # Exception number.
1410145     jmp exception_unrecoverable
1410146 #-----
1410147 isr_9:          # «coprocessor segment overrun
1410148                # exception»
1410149     cli
1410150     push $0      # Null error code.
1410151     push $9      # Exception number.
1410152     jmp exception_unrecoverable
1410153 #-----
1410154 isr_10:         # «bad TSS exception»
1410155     cli
1410156     #           # Error code already present.
1410157     push $10     # Exception number.
1410158     jmp exception_unrecoverable
1410159 #-----
1410160 isr_11:         # «segment not present exception»
1410161     cli
1410162     #           # Error code already present.
1410163     push $11     # Exception number.
1410164     jmp exception_unrecoverable
```

```
1410165 #-----
1410166 isr_12:      # «stack fault exception»
1410167     cli
1410168     #          # Error code already present.
1410169     push $12   # Exception number.
1410170     jmp exception_unrecoverable
1410171 #-----
1410172 isr_13:      # «general protection fault exception»
1410173     cli
1410174     #          # Error code already present.
1410175     push $13   # Exception number.
1410176     jmp exception_unrecoverable
1410177 #-----
1410178 isr_14:      # «page fault exception»
1410179     cli
1410180     #          # Error code already present.
1410181     push $14   # Exception number.
1410182     jmp exception_unrecoverable
1410183 #-----
1410184 isr_15:      # «unknown interrupt exception»
1410185     cli
1410186     push $0    # Null error code.
1410187     push $15   # Exception number.
1410188     jmp exception_unrecoverable
1410189 #-----
1410190 isr_16:      # «coprocessor fault exception»
1410191     cli
1410192     push $0    # Null error code.
1410193     push $16   # Exception number.
1410194     jmp exception_unrecoverable
1410195 #-----
1410196 isr_17:      # «alignment check exception»
1410197     cli
1410198     push $0    # Null error code.
1410199     push $17   # Exception number.
1410200     jmp exception_unrecoverable
1410201 #-----
```

```
1410202  isr_18:          # «machine check exception»
1410203      cli
1410204      push $0      # Null error code.
1410205      push $18     # Exception number.
1410206      jmp exception_unrecoverable
1410207  #-----
1410208  isr_19:          # «reserved exception»
1410209      cli
1410210      push $0      # Null error code.
1410211      push $19     # Exception number.
1410212      jmp exception_unrecoverable
1410213  #-----
1410214  isr_20:          # «reserved exception»
1410215      cli
1410216      push $0      # Null error code.
1410217      push $20     # Exception number.
1410218      jmp exception_unrecoverable
1410219  #-----
1410220  isr_21:          # «reserved exception»
1410221      cli
1410222      push $0      # Null error code.
1410223      push $21     # Exception number.
1410224      jmp exception_unrecoverable
1410225  #-----
1410226  isr_22:          # «reserved exception»
1410227      cli
1410228      push $0      # Null error code.
1410229      push $22     # Exception number.
1410230      jmp exception_unrecoverable
1410231  #-----
1410232  isr_23:          # «reserved exception»
1410233      cli
1410234      push $0      # Null error code.
1410235      push $23     # Exception number.
1410236      jmp exception_unrecoverable
1410237  #-----
1410238  isr_24:          # «reserved exception»
```

```
1410239     cli
1410240     push $0      # Null error code.
1410241     push $24    # Exception number.
1410242     jmp exception_unrecoverable
1410243 #-----
1410244 isr_25:      # «reserved exception»
1410245     cli
1410246     push $0      # Null error code.
1410247     push $25    # Exception number.
1410248     jmp exception_unrecoverable
1410249 #-----
1410250 isr_26:      # «reserved exception»
1410251     cli
1410252     push $0      # Null error code.
1410253     push $26    # Exception number.
1410254     jmp exception_unrecoverable
1410255 #-----
1410256 isr_27:      # «reserved exception»
1410257     cli
1410258     push $0      # Null error code.
1410259     push $27    # Exception number.
1410260     jmp exception_unrecoverable
1410261 #-----
1410262 isr_28:      # «reserved exception»
1410263     cli
1410264     push $0      # Null error code.
1410265     push $28    # Exception number.
1410266     jmp exception_unrecoverable
1410267 #-----
1410268 isr_29:      # «reserved exception»
1410269     cli
1410270     push $0      # Null error code.
1410271     push $29    # Exception number.
1410272     jmp exception_unrecoverable
1410273 #-----
1410274 isr_30:      # «reserved exception»
1410275     cli
```

```
1410276     push $0      # Null error code.
1410277     push $30    # Exception number.
1410278     jmp exception_unrecoverable
1410279 #-----
1410280 isr_31:      # «reserved exception»
1410281     cli
1410282     push $0    # Null error code.
1410283     push $31   # Exception number.
1410284     jmp exception_unrecoverable
1410285 #-----
1410286 isr_32:      # IRQ 0: «timer»
1410287     cli
1410288     jmp irq_timer
1410289 #-----
1410290 isr_33:      # IRQ 1: tastiera
1410291     cli
1410292     jmp irq_keyboard
1410293 #-----
1410294 isr_34:      # IRQ 2: it is fired for IRQ 8 to 15.
1410295     cli
1410296     #
1410297     # IRQ 2 must be ON inside the file
1410298     # 'kernel/proc/proc_init.c', so
1410299     # that it is guaranteed that the PIC 1 is reset
1410300     # here.
1410301     #
1410302     call isr_irq_clear_pic1
1410303     #
1410304     # For IRQ 2 there is nothing else to do, because it
1410305     # is a link to the PIC 2 (IRQ 8 to 15).
1410306     #
1410307     iret
1410308 #-----
1410309 isr_35:      # IRQ 3
1410310     cli
1410311     jmp irq_pic1
1410312 #-----
```

```
1410313 isr_36:          # IRQ 4
1410314     cli
1410315     jmp irq_pic1
1410316 #-----
1410317 isr_37:          # IRQ 5
1410318     cli
1410319     jmp irq_pic1
1410320 #-----
1410321 isr_38:          # IRQ 6: floppy disk drive
1410322     cli
1410323     jmp irq_pic1
1410324 #-----
1410325 isr_39:          # IRQ 7: LPT 1
1410326     cli
1410327     jmp irq_pic1
1410328 #-----
1410329 isr_40:          # IRQ 8: «real time clock (RTC)»
1410330     cli
1410331     jmp irq_pic2
1410332 #-----
1410333 isr_41:          # IRQ 9
1410334     cli
1410335     jmp irq_pic2
1410336 #-----
1410337 isr_42:          # IRQ 10
1410338     cli
1410339     jmp irq_pic2
1410340 #-----
1410341 isr_43:          # IRQ 11
1410342     cli
1410343     jmp irq_pic2
1410344 #-----
1410345 isr_44:          # IRQ 12: mouse PS/2
1410346     cli
1410347     jmp irq_pic2
1410348 #-----
1410349 isr_45:          # IRQ 13: math coprocessor
```



```
1410350     cli
1410351     jmp irq_pic2
1410352 #-----
1410353 isr_46:      # IRQ 14: primary IDE channel
1410354     cli
1410355     jmp irq_pic2
1410356 #-----
1410357 isr_47:      # IRQ 15: secondary IDE channel
1410358     cli
1410359     jmp irq_pic2
1410360 #-----
1410361 #
1410362 # Unrecoverable exceptions.
1410363 #
1410364 exception_unrecoverable:
1410365     #
1410366     # Previous pushes:
1410367     # [omissis]
1410368     # push %eflags
1410369     # push %cs
1410370     # push %eip
1410371     #
1410372     # push $<error_number>
1410373     # push $<idt_item_number>
1410374     #
1410375     pushl %gs
1410376     pushl %fs
1410377     pushl %es
1410378     pushl %ds
1410379     pushl %edi
1410380     pushl %esi
1410381     pushl %ebp
1410382     pushl %ebx
1410383     pushl %edx
1410384     pushl %ecx
1410385     pushl %eax
1410386     #
```

```
1410387     call isr_exception_unrecoverable
1410388     #
1410389     popl %eax
1410390     popl %ecx
1410391     popl %edx
1410392     popl %ebx
1410393     popl %ebp
1410394     popl %esi
1410395     popl %edi
1410396     popl %ds
1410397     popl %es
1410398     popl %fs
1410399     popl %gs
1410400     #
1410401     # Remove the IDT item number and the error code.
1410402     #
1410403     add $4, %esp
1410404     add $4, %esp
1410405     #
1410406     # Return from interrupt.
1410407     #
1410408     iret
1410409     #-----
1410410     #
1410411     # Unspecified IRQ. Currently unused.
1410412     #
1410413     irq:
1410414     #
1410415     # Previous pushes:
1410416     # [omissis]
1410417     # push %eflags
1410418     # push %cs
1410419     # push %eip
1410420     #
1410421     # push $0
1410422     # push $<idt_item_number>
1410423     #
```

```
1410424     call isr_irq_clear
1410425     #
1410426     # Remove the IDT item number and the null error
1410427     # code.
1410428     #
1410429     add $4, %esp
1410430     add $4, %esp
1410431     #
1410432     # Return from interrupt.
1410433     #
1410434     iret
1410435 #-----
1410436 #
1410437 # Keyboard IRQ.
1410438 #
1410439 irq_keyboard:
1410440     #
1410441     # Previous pushes:
1410442     # [omissis]
1410443     # push %eflags
1410444     # push %cs
1410445     # push %eip
1410446     #
1410447     pushl %gs
1410448     pushl %fs
1410449     pushl %es
1410450     pushl %ds
1410451     pushl %edi
1410452     pushl %esi
1410453     pushl %ebp
1410454     pushl %ebx
1410455     pushl %edx
1410456     pushl %ecx
1410457     pushl %eax
1410458     #
1410459     # Set the data segments to the kernel data segment,
1410460     # so that the following variables can be accessed.
```

```
1410461      #
1410462      mov  $16, %ax # DS, ES, FS and GS.
1410463      mov  %ax, %ds
1410464      mov  %ax, %es
1410465      mov  %ax, %fs
1410466      mov  %ax, %gs
1410467      #
1410468      # Check if a system call is already working: if so,
1410469      # just leave (go to L1).
1410470      #
1410471      cmpl $1, syscall_working
1410472      je  L1
1410473      #
1410474      # Call the keyboard handler.
1410475      #
1410476      call kbd_isr
1410477      #
1410478 L1: # Restore original registers and return.
1410479      #
1410480      jmp irq_pic1_pop_iret
1410481      #-----
1410482      #
1410483      # Generic IRQ from PIC 1
1410484      #
1410485      irq_pic1:
1410486      #
1410487      # Previous pushes:
1410488      # [omissis]
1410489      # push %eflags
1410490      # push %cs
1410491      # push %eip
1410492      #
1410493      pushl %gs
1410494      pushl %fs
1410495      pushl %es
1410496      pushl %ds
1410497      pushl %edi
```

```
1410498     pushl %esi
1410499     pushl %ebp
1410500     pushl %ebx
1410501     pushl %edx
1410502     pushl %ecx
1410503     pushl %eax
1410504     #
1410505     # Set the data segments to the kernel data segment,
1410506     # so that the following variables can be accessed.
1410507     #
1410508     mov  $16, %ax # DS, ES, FS and GS.
1410509     mov  %ax, %ds
1410510     mov  %ax, %es
1410511     mov  %ax, %fs
1410512     mov  %ax, %gs
1410513     #
1410514     # Check if a system call is already working: if so,
1410515     # just leave (go to L2).
1410516     #
1410517     cmpl $1, syscall_working
1410518     je  L2
1410519     #
1410520     # If we are here, no system call is working and a
1410521     # user process was interrupted.
1410522     # Save process stack registers into kernel data
1410523     # segment.
1410524     #
1410525     mov %ss,  proc_stack_segment_selector
1410526     mov %esp, proc_stack_pointer
1410527     #
1410528     # Check if it is already in kernel mode: the kernel
1410529     # has PID 0.
1410530     # If so, just leave (go to L2).
1410531     #
1410532     mov proc_current, %edx          # Interrupted PID.
1410533     mov $0,           %eax          # Kernel PID.
1410534     cmp %eax, %edx
```

```
1410535     je L5
1410536     #
1410537     # If we are here, a user process was interrupted.
1410538     # Switch to the kernel stack (data segment
1410539     # descriptor).
1410540     #
1410541     mov $16, %ax
1410542     mov %ax, %ss
1410543     mov _ksp, %esp
1410544     #
1410545     # Call the scheduler.
1410546     #
1410547     call proc_scheduler
1410548     #
1410549     # Restore process stack registers from kernel data
1410550     # segment.
1410551     #
1410552     mov proc_stack_segment_selector, %ss
1410553     mov proc_stack_pointer, %esp
1410554     #
1410555 L5:    # Restore from process stack and return.
1410556     #
1410557     jmp irq_pic1_pop_iret
1410558     #-----
1410559     #
1410560     # Generic IRQ from PIC 2
1410561     #
1410562 irq_pic2:
1410563     #
1410564     # Previous pushes:
1410565     # [omissis]
1410566     # push %eflags
1410567     # push %cs
1410568     # push %eip
1410569     #
1410570     pushl %gs
1410571     pushl %fs
```

```
1410572     pushl %es
1410573     pushl %ds
1410574     pushl %edi
1410575     pushl %esi
1410576     pushl %ebp
1410577     pushl %ebx
1410578     pushl %edx
1410579     pushl %ecx
1410580     pushl %eax
1410581     #
1410582     # Set the data segments to the kernel data segment,
1410583     # so that the following variables can be accessed.
1410584     #
1410585     mov  $16, %ax # DS, ES, FS and GS.
1410586     mov  %ax, %ds
1410587     mov  %ax, %es
1410588     mov  %ax, %fs
1410589     mov  %ax, %gs
1410590     #
1410591     # Check if a system call is already working: if so,
1410592     # just leave (go to L2).
1410593     #
1410594     cmpl $1, syscall_working
1410595     je  L2
1410596     #
1410597     # If we are here, no system call is working and a
1410598     # user process was interrupted.
1410599     # Save process stack registers into kernel data
1410600     # segment.
1410601     #
1410602     mov %ss,  proc_stack_segment_selector
1410603     mov %esp, proc_stack_pointer
1410604     #
1410605     # Check if it is already in kernel mode: the kernel
1410606     # has PID 0.
1410607     # If so, just leave (go to L2).
1410608     #
```

```
1410609     mov proc_current, %edx           # Interrupted PID.
1410610     mov $0,             %eax         # Kernel PID.
1410611     cmp %eax, %edx
1410612     je L4
1410613     #
1410614     # If we are here, a user process was interrupted.
1410615     # Switch to the kernel stack (data segment
1410616     # descriptor).
1410617     #
1410618     mov $16, %ax
1410619     mov %ax, %ss
1410620     mov _ksp, %esp
1410621     #
1410622     # Call the scheduler.
1410623     #
1410624     call proc_scheduler
1410625     #
1410626     # Restore process stack registers from kernel data
1410627     # segment.
1410628     #
1410629     mov proc_stack_segment_selector, %ss
1410630     mov proc_stack_pointer, %esp
1410631     #
1410632 L4: # Restore from process stack and return.
1410633     #
1410634     jmp irq_pic2_pop_iret
1410635     #-----
1410636     #
1410637     # Timer IRQ.
1410638     #
1410639 irq_timer:
1410640     #
1410641     # Previous pushes:
1410642     # [omissis]
1410643     # push %eflags
1410644     # push %cs
1410645     # push %eip
```



```
1410646      #
1410647      pushl %gs
1410648      pushl %fs
1410649      pushl %es
1410650      pushl %ds
1410651      pushl %edi
1410652      pushl %esi
1410653      pushl %ebp
1410654      pushl %ebx
1410655      pushl %edx
1410656      pushl %ecx
1410657      pushl %eax
1410658      #
1410659      # Set the data segments to the kernel data segment,
1410660      # so that the following variables can be accessed.
1410661      #
1410662      mov  $16, %ax # DS, ES, FS and GS.
1410663      mov  %ax, %ds
1410664      mov  %ax, %es
1410665      mov  %ax, %fs
1410666      mov  %ax, %gs
1410667      #
1410668      # Increment time counters, to keep time.
1410669      #
1410670      add $1, kticks_lo      # Kernel ticks counter.
1410671      adc $0, kticks_hi      #
1410672      #
1410673      add $1, tticks_lo      # Clock ticks counter.
1410674      adc $0, tticks_hi      #
1410675      #
1410676      # Check if a system call is already working: if so,
1410677      # just leave (go to L2).
1410678      #
1410679      cmpl $1, syscall_working
1410680      je  L2
1410681      #
1410682      # If we are here, no system call is working and a
```

```
1410683     # user process was interrupted.
1410684     # Save process stack registers into kernel data
1410685     # segment.
1410686     #
1410687     mov %ss,  proc_stack_segment_selector
1410688     mov %esp, proc_stack_pointer
1410689     #
1410690     # Check if it is already in kernel mode: the kernel
1410691     # has PID 0.
1410692     # If so, just leave (go to L2).
1410693     #
1410694     mov proc_current, %edx          # Interrupted PID.
1410695     mov $0,          %eax          # Kernel PID.
1410696     cmp %eax, %edx
1410697     je L2
1410698     #
1410699     # If we are here, a user process was interrupted.
1410700     # Switch to the kernel stack (data segment
1410701     # descriptor).
1410702     #
1410703     mov $16, %ax
1410704     mov %ax, %ss
1410705     mov _ksp, %esp
1410706     #
1410707     # Call the scheduler.
1410708     #
1410709     call proc_scheduler
1410710     #
1410711     # Restore process stack registers from kernel data
1410712     # segment.
1410713     #
1410714     mov proc_stack_segment_selector, %ss
1410715     mov proc_stack_pointer, %esp
1410716     #
1410717 L2: # Restore from process stack and return.
1410718     #
1410719     jmp irq_pic1_pop_iret
```

```
1410720 #-----
1410721 irq_pic1_pop_iret:
1410722 #
1410723 # Restore from process stack.
1410724 #
1410725 popl %eax
1410726 popl %ecx
1410727 popl %edx
1410728 popl %ebx
1410729 popl %ebp
1410730 popl %esi
1410731 popl %edi
1410732 popl %ds
1410733 popl %es
1410734 popl %fs
1410735 popl %gs
1410736 #
1410737 # End of hardware interrupt to PIC 1.
1410738 #
1410739 call isr_irq_clear_pic1
1410740 #
1410741 # Return from interrupt.
1410742 #
1410743 iret
1410744 #-----
1410745 irq_pic2_pop_iret:
1410746 #
1410747 # Restore from process stack.
1410748 #
1410749 popl %eax
1410750 popl %ecx
1410751 popl %edx
1410752 popl %ebx
1410753 popl %ebp
1410754 popl %esi
1410755 popl %edi
1410756 popl %ds
```

```
1410757     popl %es
1410758     popl %fs
1410759     popl %gs
1410760     #
1410761     # End of hardware interrupt to PIC 2 and PIC1.
1410762     #
1410763     call isr_irq_clear_pic2
1410764     #
1410765     # Return from interrupt.
1410766     #
1410767     iret
1410768 #-----
1410769 #
1410770 # System call.
1410771 #
1410772 isr_128:
1410773     #
1410774     # Previous pushes:
1410775     # [omissis]
1410776     # push message_size
1410777     # push &message_structure
1410778     # push syscall_number
1410779     # push back_address      # made by a call to sys()
1410780     # push %eflags           # made by int $128 (0x80)
1410781     # push %cs                # made by int $128 (0x80)
1410782     # push %eip               # made by int $128 (0x80)
1410783     #
1410784 #-----
1410785 #
1410786 # Save into process stack:
1410787 #
1410788     pushl %gs
1410789     pushl %fs
1410790     pushl %es
1410791     pushl %ds
1410792     pushl %edi
1410793     pushl %esi
```

```
1410794     pushl %ebp
1410795     pushl %ebx
1410796     pushl %edx
1410797     pushl %ecx
1410798     pushl %eax
1410799     #
1410800     # Set the data segments to the kernel data segment.
1410801     #
1410802     mov  $16, %ax # DS, ES, FS and GS.
1410803     mov  %ax, %ds
1410804     mov  %ax, %es
1410805     mov  %ax, %fs
1410806     mov  %ax, %gs
1410807     #
1410808     # Tell that it is a system call.
1410809     #
1410810     movl $1, syscall_working
1410811     #
1410812     # Save process stack registers into kernel data
1410813     # segment.
1410814     #
1410815     mov %ss,  proc_stack_segment_selector
1410816     mov %esp, proc_stack_pointer
1410817     #
1410818     # Save some more data, from the system call.
1410819     #
1410820     .equ SYSCALL_NUMBER,      60
1410821     .equ MESSAGE_OFFSET,     64
1410822     .equ MESSAGE_SIZE,      68
1410823     #
1410824     mov %esp, %ebp
1410825     mov SYSCALL_NUMBER(%ebp), %eax
1410826     mov %eax, proc_syscallnr
1410827     mov MESSAGE_OFFSET(%ebp), %eax
1410828     mov %eax, proc_msg_offset
1410829     mov MESSAGE_SIZE(%ebp), %eax
1410830     mov %eax, proc_msg_size
```

```
1410831      #
1410832      # Check if it is already in kernel mode: the kernel
1410833      # has PID 0.
1410834      #
1410835      mov proc_current, %edx      # Interrupted PID.
1410836      mov $0,                %eax      # Kernel PID.
1410837      cmp %eax, %edx
1410838      jne L3
1410839      #
1410840      # It is already the kernel stack, so, the variable
1410841      # "_ksp" is aligned to current stack pointer.
1410842      # This way, the first syscall
1410843      # can work without having to set the "_ksp"
1410844      # variable to some reasonable value.
1410845      #
1410846      mov %esp, _ksp
1410847      #
1410848 L3:      # Switch to the kernel stack (data segment
1410849      # descriptor).
1410850      #
1410851      mov $16, %ax
1410852      mov %ax, %ss
1410853      mov _ksp, %esp
1410854      #
1410855      # Call the external sysroutine handler.
1410856      #
1410857      push proc_msg_size
1410858      push proc_msg_offset
1410859      push proc_syscallnr
1410860      call sysroutine
1410861      add $4, %esp
1410862      add $4, %esp
1410863      add $4, %esp
1410864      #
1410865      # Restore process stack registers from kernel data
1410866      # segment.
1410867      #
```

```
1410868     mov proc_stack_segment_selector, %ss
1410869     mov proc_stack_pointer, %esp
1410870     #
1410871     # End of system call.
1410872     #
1410873     movl $0, syscall_working
1410874     #
1410875     # Restore from process stack.
1410876     #
1410877     popl %eax
1410878     popl %ecx
1410879     popl %edx
1410880     popl %ebx
1410881     popl %ebp
1410882     popl %esi
1410883     popl %edi
1410884     popl %ds
1410885     popl %es
1410886     popl %fs
1410887     popl %gs
1410888     #
1410889     # Return from interrupt.
1410890     #
1410891     iret
1410892     #-----
```

94.6.22 kernel/ibm_i386/isr_exception_name.c

Si veda la sezione [93.7](#).

```
1420001     #include <kernel/ibm_i386.h>
1420002     //-----
1420003     char *
1420004     isr_exception_name (int exception)
1420005     {
1420006         char *description[19] = { "division by zero",
1420007             "debug",
```



```

1420008     "non maskable interrupt",
1420009     "breakpoint",
1420010     "into detected overflow",
1420011     "out of bounds",
1420012     "invalid opcode",
1420013     "no coprocessor",
1420014     "double fault",
1420015     "coprocessor segmento overrun",
1420016     "bad TSS",
1420017     "segment not present",
1420018     "stack fault",
1420019     "general protection fault",
1420020     "page fault",
1420021     "unknown interrupt",
1420022     "coprocessor fault",
1420023     "alignment check",
1420024     "machine check"
1420025 };
1420026 //
1420027 if (exception >= 0 && exception <= 18)
1420028     {
1420029         return description[exception];
1420030     }
1420031 else
1420032     {
1420033         return "unknown";
1420034     }
1420035 }

```

94.6.23 kernel/ibm_i386/isr_exception_unrecoverable.c



Si veda la sezione [93.7](#).

```

1430001 #include <kernel/ibm_i386.h>
1430002 #include <kernel/lib_k.h>
1430003 #include <sys/types.h>
1430004 //-----

```



```
1430005 void
1430006 isr_exception_unrecoverable (uint32_t eax,
1430007                               uint32_t ecx,
1430008                               uint32_t edx,
1430009                               uint32_t ebx,
1430010                               uint32_t ebp,
1430011                               uint32_t esi,
1430012                               uint32_t edi, uint32_t ds,
1430013                               uint32_t es, uint32_t fs,
1430014                               uint32_t gs,
1430015                               uint32_t interrupt,
1430016                               uint32_t error,
1430017                               uint32_t eip, uint32_t cs,
1430018                               uint32_t eflags)
1430019 {
1430020     pid_t pid = cs / 8;
1430021     //
1430022     k_printf
1430023         ("[%s] ERROR: pid: %i exception %i: \"%s\"\n",
1430024          __func__, pid, interrupt,
1430025          isr_exception_name (interrupt));
1430026     //
1430027     // Exit the unrecoverable application.
1430028     //
1430029     k_exit ();
1430030 }
```

94.6.24 kernel/ibm_i386/isr_irq_clear.c

Si veda la sezione [93.7](#).

```
1440001 #include <kernel/ibm_i386.h>
1440002 //-----
1440003 void
1440004 isr_irq_clear (uint32_t idtn)
1440005 {
1440006     int irq = idtn - 32;
```

```
1440007 //
1440008 // Must tell the PIC (programmable interrupt
1440009 // controller).
1440010 //
1440011 // If the IRQ number is between 8 and 15, send
1440012 // message «EOI»
1440013 // (End of IRQ) to PIC 2.
1440014 //
1440015 if (irq >= 8)
1440016 {
1440017     out_8 (0xA0, 0x20);
1440018 }
1440019 //
1440020 // Then send message «EOI» to PIC 1.
1440021 //
1440022 out_8 (0x20, 0x20);
1440023 }
```

94.6.25 kernel/ibm_i386/isr_irq_clear_pic1.c

«

Si veda la sezione [93.7](#).

```
1450001 #include <kernel/ibm_i386.h>
1450002 //-----
1450003 void
1450004 isr_irq_clear_pic1 (void)
1450005 {
1450006     //
1450007     // Send message «EOI» to PIC 1.
1450008     //
1450009     out_8 ((uint32_t) 0x20, (uint32_t) 0x20);
1450010 }
```

94.6.26 kernel/ibm_i386/isr_irq_clear_pic2.c



Si veda la sezione [93.7](#).

```
1460001 #include <kernel/ibm_i386.h>
1460002 //-----
1460003 void
1460004 isr_irq_clear_pic2 (void)
1460005 {
1460006     //
1460007     // Send message «EOI» (End of IRQ) to PIC 2.
1460008     // It must be sent after a IRQ number between 8 and
1460009     // 15, but after
1460010     // that, remember that also the PIC 1 must receive
1460011     // an «EOI» message
1460012     // (maybe with the help of 'isr_irq_clear_pic1()'
1460013     // function).
1460014     //
1460015     out_8 ((uint32_t) 0xA0, (uint32_t) 0x20);
1460016 }
```

94.6.27 kernel/ibm_i386/sti.s



Si veda la sezione [93.7](#).

```
1470001 .global sti
1470002 #-----
1470003 .text
1470004 #-----
1470005 # Set interrupt flag.
1470006 #-----
1470007 .align 4
1470008 sti:
1470009     sti
1470010     ret
```

94.7 os32: «kernel/lib_k.h»



Si veda la sezione [93.11](#).

```

1480001 #ifndef _KERNEL_LIB_K_H
1480002 #define _KERNEL_LIB_K_H          1
1480003 //-----
1480004 #include <restrict.h>
1480005 #include <size_t.h>
1480006 #include <stdarg.h>
1480007 #include <stdint.h>
1480008 #include <time.h>
1480009 #include <unistd.h>
1480010 #include <kernel/lib_s.h>
1480011 //-----
1480012 void k_exit (void);
1480013 unsigned int k_sleep (unsigned int seconds);
1480014 int k_usleep (useconds_t usec);
1480015 char *k_gets (char *s);
1480016 void k_perror (const char *s);
1480017 int k_printf (const char *restrict format, ...);
1480018 int k_stime (time_t * timer);
1480019 int k_vprintf (const char *restrict format, va_list arg);
1480020 int k_vsprintf (char *restrict string,
1480021                const char *restrict format, va_list arg);
1480022 //clock_t      k_clock      (void);
1480023 #define      k_clock()      (s_clock ((uid_t) 0))
1480024 #define      k_time(t)      (s_time ((pid_t) 0, t))
1480025 //-----
1480026 #endif

```

94.7.1	kernel/lib_k/k_exit.s	1333
94.7.2	kernel/lib_k/k_gets.c	1333
94.7.3	kernel/lib_k/k_perror.c	1334
94.7.4	kernel/lib_k/k_printf.c	1335

Script e sorgenti del kernel	1333
94.7.5 kernel/lib_k/k_sleep.c	1336
94.7.6 kernel/lib_k/k_stime.c	1337
94.7.7 kernel/lib_k/k_usleep.c	1337
94.7.8 kernel/lib_k/k_vprintf.c	1339
94.7.9 kernel/lib_k/k_vsprintf.c	1340

94.7.1 kernel/lib_k/k_exit.s



Si veda la sezione [93.11](#).

```

1490001  .global k_exit
1490002  #-----
1490003  .text
1490004  #-----
1490005  .align 4
1490006  k_exit:
1490007  halt:
1490008      hlt
1490009      jmp halt

```

94.7.2 kernel/lib_k/k_gets.c



Si veda la sezione [93.11](#).

```

1500001  #include <kernel/lib_k.h>
1500002  #include <kernel/driver/kbd.h>
1500003  //-----
1500004  char *
1500005  k_gets (char *s)
1500006  {
1500007      int i;
1500008      //
1500009      // Legge kbd.char.
1500010      //

```

```
1500011     for (i = 0; i < 256; i++)
1500012     {
1500013         while (kbd.key == 0)
1500014         {
1500015             //
1500016             // Attende un carattere.
1500017             //
1500018             ;
1500019         }
1500020         s[i] = kbd.key;
1500021         kbd.key = 0;
1500022         if (s[i] == '\n')
1500023         {
1500024             s[i] = 0;
1500025             break;
1500026         }
1500027     }
1500028     return s;
1500029 }
```

94.7.3 kernel/lib_k/k_perror.c

<<

Si veda la sezione [93.11](#).

```
1510001 #include <kernel/lib_k.h>
1510002 #include <errno.h>
1510003 //-----
1510004 void
1510005 k_perror (const char *s)
1510006 {
1510007     //
1510008     // If errno is zero, there is nothing to show.
1510009     //
1510010     if (errno == 0)
1510011     {
1510012         return;
1510013     }
```

```

1510014 //
1510015 // Show the string if there is one.
1510016 //
1510017 if (s != NULL && strlen (s) > 0)
1510018 {
1510019     k_printf ("%s: ", s);
1510020 }
1510021 //
1510022 // Show the translated error.
1510023 //
1510024 if (errfn[0] != 0 && errln != 0)
1510025 {
1510026     k_printf ("%s:%u:%i] %s\n",
1510027               errfn, errln, errno, strerror (errno));
1510028 }
1510029 else
1510030 {
1510031     k_printf ("%i] %s\n", errno, strerror (errno));
1510032 }
1510033 }

```

94.7.4 kernel/lib_k/k_printf.c

Si veda la sezione [93.11](#).

```

1520001 #include <stdarg.h>
1520002 #include <kernel/lib_k.h>
1520003 //-----
1520004 int
1520005 k_printf (const char *restrict format, ...)
1520006 {
1520007     va_list ap;
1520008     va_start (ap, format);
1520009     return k_vprintf (format, ap);
1520010 }

```



94.7.5 kernel/lib_k/k_sleep.c



Si veda la sezione [93.11](#).

```
1530001 #include <kernel/lib_k.h>
1530002 #include <kernel/lib_s.h>
1530003 #include <kernel/proc.h>
1530004 #include <time.h>
1530005 //-----
1530006 unsigned int
1530007 k_sleep (unsigned int seconds)
1530008 {
1530009     clock_t time_start;
1530010     clock_t time_now;
1530011     clock_t time_elapsed;
1530012     clock_t time = seconds * CLOCKS_PER_SEC;
1530013     unsigned long long int loops;
1530014     //
1530015     // Calculate how many times the following loop
1530016     // should
1530017     // be run to get the requested time.
1530018     //
1530019     loops = proc_loops_per_clock * time;
1530020     //
1530021     // Do a wasting time loop.
1530022     //
1530023     time_elapsed = 0;
1530024     time_start = s_clock ((pid_t) 0);
1530025     //
1530026     // If the function 's_clock()' can help, it will
1530027     // exit even if the loop is become slow, but the
1530028     // time was correctly accounted and is elapsed.
1530029     //
1530030     for (; time_elapsed < time && loops > 0; loops--)
1530031     {
1530032         time_now = s_clock ((pid_t) 0);
1530033         time_elapsed = time_now - time_start;
1530034     }
```



```

1530035 //
1530036 // The sleep is always complete.
1530037 //
1530038 return (0);
1530039 }

```

94.7.6 kernel/lib_k/k_stime.c



Si veda la sezione [93.11](#).

```

1540001 #include <kernel/lib_k.h>
1540002 //-----
1540003 extern clock_t _clock_time; // uint64_t
1540004 //-----
1540005 int
1540006 k_stime (time_t * timer)
1540007 {
1540008     _clock_time = (*timer * CLOCKS_PER_SEC);
1540009     return (0);
1540010 }

```

94.7.7 kernel/lib_k/k_usleep.c



Si veda la sezione [93.11](#).

```

1550001 #include <kernel/lib_k.h>
1550002 #include <kernel/lib_s.h>
1550003 #include <kernel/proc.h>
1550004 #include <time.h>
1550005 #include <unistd.h>
1550006 //-----
1550007 int
1550008 k_usleep (useconds_t usec)
1550009 {
1550010     clock_t time_start;
1550011     clock_t time_now;
1550012     clock_t time_elapsed;

```

```
1550013 clock_t time;
1550014 unsigned long long int loops;
1550015 //
1550016 // Calculate time, in terms of internal clocks
1550017 //
1550018 if (usec < 10000000)
1550019 {
1550020     time = (usec * CLOCKS_PER_SEC) / 1000000;
1550021 }
1550022 else
1550023 {
1550024     time = (usec / 1000000) * CLOCKS_PER_SEC;
1550025 }
1550026 //
1550027 // Fix time: if it is zero, it means that it was
1550028 // requested a sleep
1550029 // shorter than the internal clock timer impulse.
1550030 // So, if it is zero,
1550031 // correct to at least a one.
1550032 //
1550033 if (time == 0 && usec != 0)
1550034     time = 1;
1550035 //
1550036 // Calculate how many times the following loop
1550037 // should
1550038 // be run to get the requested time.
1550039 //
1550040 loops = proc_loops_per_clock * time;
1550041 //
1550042 // Do a wasting time loop.
1550043 //
1550044 time_elapsed = 0;
1550045 time_start = s_clock ((pid_t) 0);
1550046 //
1550047 // If the function 's_clock()' can help, it will
1550048 // exit even if the loop is become slow, but the
1550049 // time was correctly accounted and is elapsed.
```

```
1550050 //
1550051 for (; time_elapsed < time && loops > 0; loops--)
1550052 {
1550053     time_now = s_clock ((pid_t) 0);
1550054     time_elapsed = time_now - time_start;
1550055 }
1550056 //
1550057 // The sleep is always complete.
1550058 //
1550059 return (0);
1550060 }
```

94.7.8 kernel/lib_k/k_vprintf.c

<<

Si veda la sezione [93.11](#).

```
1560001 #include <kernel/lib_k.h>
1560002 #include <stdio.h>
1560003 #include <kernel/dev.h>
1560004 //-----
1560005 int
1560006 k_vprintf (const char *restrict format, va_list arg)
1560007 {
1560008     size_t size = BUFSIZ;
1560009     char string[BUFSIZ];
1560010     int status;
1560011     //
1560012     // At the moment, set 'string' to be a null string.
1560013     //
1560014     string[0] = 0;
1560015     //
1560016     // Get the string, that must be limited to 'size'
1560017     // bytes.
1560018     //
1560019     status = vsnprintf (string, size, format, arg);
1560020     //
1560021     //
```

```
1560022 //
1560023 if (status < 0)
1560024 {
1560025 //
1560026 // Variable 'errno' is not updated.
1560027 //
1560028 return (status);
1560029 }
1560030 //
1560031 // Get size.
1560032 //
1560033 size = status;
1560034 //
1560035 // Write to the first console.
1560036 //
1560037 dev_io ((pid_t) 0, DEV_CONSOLE0, DEV_WRITE,
1560038         (off_t) 0, string, size, NULL);
1560039 //
1560040 // Return the same value obtained from 'vsnprintf()'
1560041 //
1560042 return status;
1560043 }
```

94.7.9 kernel/lib_k/k_vsprintf.c



Si veda la sezione [93.11](#).

```
1570001 #include <stdarg.h>
1570002 #include <kernel/lib_k.h>
1570003 #include <stdio.h>
1570004 //-----
1570005 int
1570006 k_vsprintf (char *restrict string,
1570007             const char *restrict format, va_list arg)
1570008 {
1570009     int status;
1570010     status = vsnprintf (string, BUFSIZ, format, arg);
```

```
1570011     return status;
1570012 }
```

94.8 os32: «kernel/lib_s.h»

<<

Si veda la sezione [93.12](#).

```
1580001 #ifndef _KERNEL_LIB_S_H
1580002 #define _KERNEL_LIB_S_H 1
1580003 //-----
1580004 #include <sys/types.h>
1580005 #include <sys/stat.h>
1580006 #include <kernel/fs.h>
1580007 #include <sys/os32.h>
1580008 #include <stddef.h>
1580009 #include <stdint.h>
1580010 #include <time.h>
1580011 #include <termios.h>
1580012 #include <setjmp.h>
1580013 //-----
1580014 void s__exit (pid_t pid, int status);
1580015 int s_brk (pid_t pid, void *address);
1580016 void *s_sbrk (pid_t pid, intptr_t increment);
1580017 pid_t s_fork (pid_t ppid);
1580018 int s_kill (pid_t pid_killer, pid_t pid_target, int sig);
1580019 void s_longjmp (pid_t pid, jmp_buf env, int val);
1580020 int s_seteuid (pid_t pid, uid_t euid);
1580021 int s_setjmp (pid_t pid, jmp_buf env);
1580022 int s_setuid (pid_t pid, uid_t uid);
1580023 int s_setegid (pid_t pid, gid_t egid);
1580024 int s_setgid (pid_t pid, gid_t gid);
1580025 pid_t s_wait (pid_t pid, int *status);
1580026 sighandler_t s_signal (pid_t pid, int sig,
1580027                       sighandler_t handler,
1580028                       uintptr_t wrapper);
1580029 //-----
1580030 int s_chdir (pid_t pid, const char *path);
```

```
1580031 int s_chmod (pid_t pid, const char *path, mode_t mode);
1580032 int s_chown (pid_t pid, const char *path, uid_t uid,
1580033             gid_t gid);
1580034 int s_link (pid_t pid, const char *path_old,
1580035            const char *path_new);
1580036 int s_mkdir (pid_t pid, const char *path, mode_t mode);
1580037 int s_mknod (pid_t pid, const char *path, mode_t mode,
1580038             dev_t device);
1580039 int s_open (pid_t pid, const char *path, int oflags,
1580040            mode_t mode);
1580041 int s_stat (pid_t pid, const char *path,
1580042            struct stat *buffer);
1580043 int s_unlink (pid_t pid, const char *path);
1580044 //
1580045 int s_pipe (pid_t pid, int pipefd[2]);
1580046 //
1580047 int s_close (pid_t pid, int fdn);
1580048 int s_dup (pid_t pid, int fdn_old);
1580049 int s_dup2 (pid_t pid, int fdn_old, int fdn_new);
1580050 int s_fchmod (pid_t pid, int fdn, mode_t mode);
1580051 int s_fchown (pid_t pid, int fdn, uid_t uid, gid_t gid);
1580052 int s_fcntl (pid_t pid, int fdn, int cmd, int arg);
1580053 int s_fstat (pid_t pid, int fdn, struct stat *buffer);
1580054 off_t s_lseek (pid_t pid, int fdn, off_t offset,
1580055               int whence);
1580056 ssize_t s_read (pid_t pid, int fdn, void *buffer,
1580057                size_t count);
1580058 ssize_t s_write (pid_t pid, int fdn,
1580059                 const void *buffer, size_t count);
1580060 //
1580061 int s_mount (pid_t pid, const char *path_dev,
1580062             const char *path_mnt, int options);
1580063 int s_umount (pid_t pid, const char *path_mnt);
1580064 //-----
1580065 int s_accept (pid_t pid, int sfdn,
1580066              struct sockaddr *addr, socklen_t * addrlen);
1580067 int s_bind (pid_t pid, int sfdn,
```

```
1580068         const struct sockaddr *addr, socklen_t addrlen);
1580069 int s_connect (pid_t pid, int sfdn,
1580070             const struct sockaddr *addr,
1580071             socklen_t addrlen);
1580072 int s_listen (pid_t pid, int sfdn, int backlog);
1580073 int s_socket (pid_t pid, int family, int type,
1580074             int protocol);
1580075 ssize_t s_send (pid_t pid, int sfdn,
1580076             const void *buffer, size_t size, int flags);
1580077 ssize_t s_recv (pid_t pid, int sfdn, void *buffer,
1580078             size_t size, int flags);
1580079 ssize_t s_recvfrom (pid_t pid, int sfdn, void *buffer,
1580080             size_t length, int flags,
1580081             struct sockaddr *addrfrom,
1580082             socklen_t * addrlen);
1580083 //
1580084 int s_ipconfig (pid_t pid, int n, h_addr_t address, int m);
1580085 int s_routeadd (pid_t pid, h_addr_t destination, int m,
1580086             h_addr_t router, int device);
1580087 int s_routedel (pid_t pid, h_addr_t destination, int m);
1580088 //-----
1580089 int s_tcgetattr (pid_t pid, int fdn,
1580090             struct termios *termios_p);
1580091 int s_tcsetattr (pid_t pid, int fdn, int action,
1580092             struct termios *termios_p);
1580093 //-----
1580094 clock_t s_clock (pid_t pid);      // [p]
1580095 time_t s_time (pid_t pid, time_t * timer);      // [p]
1580096 int s_stime (pid_t pid, time_t * timer);
1580097 //
1580098 // [p] The PID information, here, is not used. The
1580099 //      argument is required only for syntax coherence
1580100 //      with other functions, that do
1580101 //      system calls, inside the kernel.
1580102 //
1580103 //-----
```

1580104	#endif
---------	--------

94.8.1	kernel/lib_s/s__exit.c	1346
94.8.2	kernel/lib_s/s_accept.c	1351
94.8.3	kernel/lib_s/s_bind.c	1355
94.8.4	kernel/lib_s/s_brk.c	1359
94.8.5	kernel/lib_s/s_chdir.c	1367
94.8.6	kernel/lib_s/s_chmod.c	1369
94.8.7	kernel/lib_s/s_chown.c	1370
94.8.8	kernel/lib_s/s_clock.c	1372
94.8.9	kernel/lib_s/s_close.c	1372
94.8.10	kernel/lib_s/s_connect.c	1375
94.8.11	kernel/lib_s/s_dup.c	1381
94.8.12	kernel/lib_s/s_dup2.c	1381
94.8.13	kernel/lib_s/s_fchmod.c	1383
94.8.14	kernel/lib_s/s_fchown.c	1384
94.8.15	kernel/lib_s/s_fcntl.c	1386
94.8.16	kernel/lib_s/s_fork.c	1388
94.8.17	kernel/lib_s/s_fstat.c	1398
94.8.18	kernel/lib_s/s_ipconfig.c	1400
94.8.19	kernel/lib_s/s_kill.c	1402
94.8.20	kernel/lib_s/s_link.c	1406

Script e sorgenti del kernel	1345
94.8.21 kernel/lib_s/s_listen.c	1408
94.8.22 kernel/lib_s/s_longjmp.c	1410
94.8.23 kernel/lib_s/s_lseek.c	1412
94.8.24 kernel/lib_s/s_mkdir.c	1414
94.8.25 kernel/lib_s/s_mknod.c	1418
94.8.26 kernel/lib_s/s_mount.c	1421
94.8.27 kernel/lib_s/s_open.c	1423
94.8.28 kernel/lib_s/s_pipe.c	1432
94.8.29 kernel/lib_s/s_read.c	1435
94.8.30 kernel/lib_s/s_recvfrom.c	1441
94.8.31 kernel/lib_s/s_routeadd.c	1456
94.8.32 kernel/lib_s/s_routedel.c	1458
94.8.33 kernel/lib_s/s_sbrk.c	1460
94.8.34 kernel/lib_s/s_send.c	1462
94.8.35 kernel/lib_s/s_setegid.c	1469
94.8.36 kernel/lib_s/s seteuid.c	1470
94.8.37 kernel/lib_s/s_setgid.c	1471
94.8.38 kernel/lib_s/s_setjmp.c	1472
94.8.39 kernel/lib_s/s_setuid.c	1474
94.8.40 kernel/lib_s/s_signal.c	1475
94.8.41 kernel/lib_s/s_socket.c	1477
94.8.42 kernel/lib_s/s_stat.c	1480

94.8.43	kernel/lib_s/s_stime.c	1483
94.8.44	kernel/lib_s/s_tcgetattr.c	1484
94.8.45	kernel/lib_s/s_tcsetattr.c	1486
94.8.46	kernel/lib_s/s_time.c	1488
94.8.47	kernel/lib_s/s_umount.c	1489
94.8.48	kernel/lib_s/s_unlink.c	1493
94.8.49	kernel/lib_s/s_wait.c	1498
94.8.50	kernel/lib_s/s_write.c	1500

94.8.1 kernel/lib_s/s__exit.c

<<

Si veda la sezione [87.2](#).

```
1590001 #include <errno.h>
1590002 #include <kernel/proc.h>
1590003 #include <kernel/lib_k.h>
1590004 #include <kernel/lib_s.h>
1590005 //-----
1590006 void
1590007 s__exit (pid_t pid, int status)
1590008 {
1590009     pid_t child;
1590010     pid_t parent = proc_table[pid].ppid;
1590011     int proc_count;
1590012     pid_t extra;
1590013     int sigchld = 0;
1590014     int fdn;
1590015     tty_t *tty;
1590016     int closed;
1590017     fd_t *fd;
1590018     //
1590019     proc_table[pid].status = PROC_ZOMBIE;
```

```
1590020     proc_table[pid].ret = status;
1590021     proc_table[pid].sig_status = 0;
1590022     proc_table[pid].sig_ignore = 0;
1590023     //
1590024     // Close files.
1590025     //
1590026     for (fdn = 0; fdn < OPEN_MAX; fdn++)
1590027     {
1590028         closed = s_close (pid, fdn);
1590029         //
1590030         // Close might fail for work in progress.
1590031         //
1590032         if (closed < 0 && errno == EINPROGRESS)
1590033         {
1590034             //
1590035             // Should be a socket, but close badly,
1590036             // because we cannot wait.
1590037             //
1590038             fd = fd_reference (pid, &fdn);
1590039             if (fd->file->sock != NULL)
1590040             {
1590041                 fd->file->sock->active = 0;
1590042                 s_close (pid, fdn);
1590043             }
1590044         }
1590045     }
1590046     //
1590047     // Close current directory.
1590048     //
1590049     inode_put (proc_table[pid].inode_cwd);
1590050     //
1590051     // Close the controlling terminal, if it is a
1590052     // process leader with
1590053     // such a terminal.
1590054     //
1590055     if (proc_table[pid].pgrp == pid
1590056         && proc_table[pid].device_tty != 0)
```

```
1590057     {
1590058         tty = tty_reference (proc_table[pid].device_tty);
1590059         //
1590060         // Verify.
1590061         //
1590062         if (tty == NULL)
1590063             {
1590064                 //
1590065                 // Show a kernel message.
1590066                 //
1590067                 k_printf
1590068                     ("kernel alert: cannot find the "
1590069                     "terminal item "
1590070                     "for device 0x%04x!\n",
1590071                     (int) proc_table[pid].device_tty);
1590072             }
1590073         else if (tty->pgrp != pid)
1590074             {
1590075                 //
1590076                 // Show a kernel message.
1590077                 //
1590078                 k_printf
1590079                     ("kernel alert: terminal "
1590080                     "device 0x%04x should "
1590081                     "be associated to the "
1590082                     "process group %i, but it "
1590083                     "is instead related to "
1590084                     "process group %i!\n",
1590085                     (int) proc_table[pid].device_tty,
1590086                     (int) pid, (int) tty->pgrp);
1590087             }
1590088         else
1590089             {
1590090                 tty->pgrp = 0;
1590091             }
1590092     }
1590093     //
```

```
1590094 // Data and text might share the same address space.
1590095 // If they are are on different places, then must
1590096 // verify if the text is not shared by other
1590097 // processes.
1590098 //
1590099 if (proc_table[pid].domain_data == 0)
1590100 {
1590101 //
1590102 // Text and data are together.
1590103 //
1590104 mb_free (proc_table[pid].address_text,
1590105          (proc_table[pid].domain_text
1590106          + proc_table[pid].extra_data));
1590107 }
1590108 else
1590109 {
1590110 //
1590111 // Data is separate and is to be removed alone.
1590112 //
1590113 mb_free (proc_table[pid].address_data,
1590114          (proc_table[pid].domain_data
1590115          + proc_table[pid].extra_data));
1590116 //
1590117 // Now must verify if no other process uses the
1590118 // same text
1590119 // memory.
1590120 //
1590121 for (proc_count = 0, extra = 0;
1590122      extra < PROCESS_MAX; extra++)
1590123 {
1590124     if (proc_table[extra].status == PROC_EMPTY ||
1590125         proc_table[extra].status == PROC_ZOMBIE)
1590126     {
1590127         continue;
1590128     }
1590129     if (proc_table[pid].address_text
1590130         == proc_table[extra].address_text)
```

```
1590131         {
1590132             proc_count++;
1590133         }
1590134     }
1590135     if (proc_count == 0)
1590136     {
1590137         //
1590138         // The code segment can be released, because
1590139         // no other process, except the current one
1590140         // (to be closed), is using it.
1590141         //
1590142         mb_free (proc_table[pid].address_text,
1590143                 proc_table[pid].domain_text);
1590144     }
1590145 }
1590146 //
1590147 // Abandon children to 'init' ((pid_t) 1).
1590148 //
1590149 for (child = 1; child < PROCESS_MAX; child++)
1590150 {
1590151     if (proc_table[child].status != PROC_EMPTY
1590152         && proc_table[child].ppid == pid)
1590153     {
1590154         proc_table[child].ppid = 1;    // Son of
1590155         // 'init'.
1590156         if (proc_table[child].status == PROC_ZOMBIE)
1590157             {
1590158                 sigchld = 1;        // Must send a SIGCHLD
1590159                 // to 'init'.
1590160             }
1590161     }
1590162 }
1590163 //
1590164 // SIGCHLD to 'init'.
1590165 //
1590166 if (sigchld
1590167     && pid != 1
```

```

1590168     && proc_table[1].status != PROC_EMPTY
1590169     && proc_table[1].status != PROC_ZOMBIE)
1590170     {
1590171         proc_sig_on ((pid_t) 1, SIGCHLD);
1590172     }
1590173     //
1590174     // Announce to the parent the death of its child.
1590175     //
1590176     if (pid != parent
1590177         && proc_table[parent].status != PROC_EMPTY)
1590178     {
1590179         proc_sig_on (parent, SIGCHLD);
1590180     }
1590181 }

```

94.8.2 kernel/lib_s/s_accept.c

Si veda la sezione [87.3](#).

```

1600001 #include <kernel/proc.h>
1600002 #include <kernel/lib_s.h>
1600003 #include <kernel/lib_k.h>
1600004 #include <errno.h>
1600005 #include <fcntl.h>
1600006 #include <sys/socket.h>
1600007 #include <arpa/inet.h>
1600008 //-----
1600009 int
1600010 s_accept (pid_t pid, int sfdn, struct sockaddr *addr,
1600011          socklen_t * addrlen)
1600012 {
1600013     fd_t *sfd;
1600014     fd_t *sfd2;
1600015     int sfdn2;
1600016     struct sockaddr_in sa;
1600017     int q;
1600018     //

```

```
1600019 // Get file descriptor and verify that it is a
1600020 // socket.
1600021 //
1600022 sfd = fd_reference (pid, &sfdn);
1600023 if (sfd == NULL || sfd->file == NULL)
1600024 {
1600025     errset (EBADF); // Bad file descriptor.
1600026     return (-1);
1600027 }
1600028 if (sfd->file->sock == NULL)
1600029 {
1600030     errset (ENOTSOCK); // Not a socket.
1600031     return (-1);
1600032 }
1600033 //
1600034 // The socket must be a stream.
1600035 //
1600036 if (sfd->file->sock->type != SOCK_STREAM)
1600037 {
1600038     errset (EOPNOTSUPP);
1600039     return (-1);
1600040 }
1600041 //
1600042 // The socket must be a TCP stream: no other stream
1600043 // types are
1600044 // available.
1600045 //
1600046 if (sfd->file->sock->protocol != IPPROTO_TCP)
1600047 {
1600048     errset (EOPNOTSUPP);
1600049     return (-1);
1600050 }
1600051 //
1600052 // The socket itself must be listening.
1600053 //
1600054 if (sfd->file->sock->tcp.conn != TCP_LISTEN)
1600055 {
```



```
1600056     errset (EINVAL);
1600057     return (-1);
1600058 }
1600059 //
1600060 // The connections must be related to the same PID.
1600061 //
1600062 if (sfd->file->sock->tcp.listen_pid != pid)
1600063 {
1600064     k_printf
1600065         ("%s] the connection pid %i is not the same "
1600066          "as the current pid %i!\n", __func__,
1600067          sfd->file->sock->tcp.listen_pid, pid);
1600068     errset (EUNKNOWN);
1600069     return (-1);
1600070 }
1600071 //
1600072 // Ok: find a connected socket from the queue.
1600073 //
1600074 for (q = 0; q < sfd->file->sock->tcp.listen_max; q++)
1600075 {
1600076     if (sfd->file->sock->tcp.listen_queue[q] != -1)
1600077     {
1600078         //
1600079         // Found.
1600080         //
1600081         break;
1600082     }
1600083 }
1600084 if (q >= sfd->file->sock->tcp.listen_max)
1600085 {
1600086     //
1600087     // At the moment, there is no new connection.
1600088     //
1600089     errset (EAGAIN); // Try again.
1600090     return (-1);
1600091 }
1600092 //
```

```
160093 // Descriptor found: check it.
160094 //
160095 sfdn2 = sfd->file->sock->tcp.listen_queue[q];
160096 sfd2 = fd_reference (pid, &sfdn2);
160097 if (sfd2 == NULL || sfd->file == NULL)
160098     {
160099         k_printf
160100             ("%s] the connected file descriptor %i "
160101             "for process %i is not a file descriptor!",
160102             __func__, sfdn2, pid);
160103         errset (EBADF); // Bad file descriptor.
160104         return (-1);
160105     }
160106 if (sfd2->file->sock == NULL)
160107     {
160108         k_printf
160109             ("%s] the connected file descriptor %i "
160110             "for process %i is not a socket!", __func__,
160111             sfdn2, pid);
160112         errset (ENOTSOCK); // Not a socket.
160113         return (-1);
160114     }
160115 //
160116 // Ok.
160117 //
160118 if (addrlen != NULL && addr != NULL && *addrlen > 0)
160119     {
160120         sa.sin_family = AF_INET;
160121         sa.sin_port = htons (sfd2->file->sock->rport);
160122         sa.sin_addr.s_addr = htonl (sfd2->file->sock->raddr);
160123         //
160124         memcpy (addr, &sa,
160125                 min (sizeof (sa), (size_t) * addrlen));
160126         *addrlen = sizeof (sa);
160127     }
160128 //
160129 // Reset the queue element and return.
```

```
1600130 //
1600131 sfd->file->sock->tcp.listen_queue[q] = -1;
1600132 return (sfdn2);
1600133 }
```

94.8.3 kernel/lib_s/s_bind.c

Si veda la sezione [87.4](#).

```
1610001 #include <kernel/proc.h>
1610002 #include <kernel/lib_s.h>
1610003 #include <kernel/lib_k.h>
1610004 #include <errno.h>
1610005 #include <fcntl.h>
1610006 #include <sys/socket.h>
1610007 #include <arpa/inet.h>
1610008 //-----
1610009 int
1610010 s_bind (pid_t pid, int sfdn,
1610011         const struct sockaddr *addr, socklen_t addrlen)
1610012 {
1610013     fd_t *sfd;
1610014     struct sockaddr_in *sin;
1610015     proc_t *ps = proc_reference (pid);
1610016     int i;
1610017     clock_t clock_time;
1610018     //
1610019     // Get file descriptor and verify that it is a
1610020     // socket.
1610021     //
1610022     sfd = fd_reference (pid, &sfdn);
1610023     if (sfd == NULL || sfd->file == NULL)
1610024     {
1610025         errset (EBADF); // Bad file descriptor.
1610026         return (-1);
1610027     }
1610028     if (sfd->file->sock == NULL)
```

```
1610029     {
1610030         errset (ENOTSOCK);           // Not a socket.
1610031         return (-1);
1610032     }
1610033     //
1610034     // Verify to have a valid address pointer.
1610035     //
1610036     if (addr == NULL)
1610037     {
1610038         errset (EINVAL);
1610039         return (-1);
1610040     }
1610041     //
1610042     // Check minimal address size.
1610043     //
1610044     if (addrlen < sizeof (struct sockaddr))
1610045     {
1610046         errset (EINVAL);
1610047         return (-1);
1610048     }
1610049     //
1610050     //
1610051     //
1610052     if (addr->sa_family == AF_INET)
1610053     {
1610054         sin = (struct sockaddr_in *) addr;
1610055         //
1610056         // The source address might be zero, to tell
1610057         // that any local
1610058         // address is valid.
1610059         //
1610060         // If it is a TCP/UDP protocol, must have valid
1610061         // ports.
1610062         //
1610063         if (sfd->file->sock->protocol == IPPROTO_TCP
1610064             || sfd->file->sock->protocol == IPPROTO_UDP)
1610065         {
```

```
1610066 //
1610067 // Local port.
1610068 //
1610069 if (ntohs (sin->sin_port) == 0)
1610070 {
1610071 //
1610072 // Missing the local port.
1610073 //
1610074 errset (EADDRNOTAVAIL);
1610075 return (-1);
1610076 }
1610077 //
1610078 // If the local port is privileged, must
1610079 // have EUID == 0.
1610080 //
1610081 if (ntohs (sin->sin_port) < 1024)
1610082 {
1610083     if (ps->euid != 0)
1610084     {
1610085 //
1610086 // Missing privileges.
1610087 //
1610088 errset (EACCES);
1610089 return (-1);
1610090     }
1610091 }
1610092 //
1610093 // If the local port is not given, a default
1610094 // one is assigned.
1610095 //
1610096 if (sfd->file->sock->lport == 0)
1610097 {
1610098 //
1610099 // Must find a free one.
1610100 //
1610101 sfd->file->sock->lport = sock_free_port ();
1610102 if (sfd->file->sock->lport == 0)
```

```
1610103         {
1610104             //
1610105             // No port is available.
1610106             //
1610107             errset (EAGAIN);
1610108             return (-1);
1610109         }
1610110     }
1610111 }
1610112 else
1610113 {
1610114     //
1610115     // Not supported.
1610116     //
1610117     errset (EOPNOTSUPP);
1610118     return (-1);
1610119 }
1610120 //
1610121 // Update the socket.
1610122 //
1610123 sfd->file->sock->family = sin->sin_family;
1610124 sfd->file->sock->laddr = ntohl (sin->sin_addr.s_addr);
1610125 sfd->file->sock->lport = ntohs (sin->sin_port);
1610126 // sfd->file->sock->bind = 1;
1610127 //
1610128 // Reset read packets clock time.
1610129 //
1610130 clock_time = s_clock (pid);
1610131 for (i = 0; i < IP_MAX_PACKETS; i++)
1610132     {
1610133         sfd->file->sock->read.clock[i] = clock_time;
1610134     }
1610135 }
1610136 else
1610137 {
1610138     //
1610139     // Unsupported family.
```

```
1610140         //
1610141         errset (EAFNOSUPPORT);
1610142         return (-1);
1610143     }
1610144     //
1610145     // Ok.
1610146     //
1610147     return (0);
1610148 }
```

94.8.4 kernel/lib_s/s_brk.c

Si veda la sezione [87.5](#).

```
1620001 #include <errno.h>
1620002 #include <kernel/proc.h>
1620003 #include <kernel/memory.h>
1620004 #include <kernel/lib_k.h>
1620005 #include <kernel/lib_s.h>
1620006 //-----
1620007 #define DEBUG 0
1620008 //-----
1620009 int
1620010 s_brk (pid_t pid, void *address)
1620011 {
1620012     size_t requested_size;
1620013     size_t requested_extra;
1620014     addr_t previous_address_text;
1620015     size_t previous_domain_text;
1620016     addr_t previous_address_data;
1620017     size_t previous_domain_data;
1620018     size_t previous_extra;
1620019     addr_t allocated_text;
1620020     addr_t allocated_data;
1620021     int status;
1620022     //
1620023     // All segments start form ((void *) 0), so the new
```

```
1620024 // address requested is equivalent to the new size
1620025 // for the data segment.
1620026 //
1620027 requested_size = (size_t) address;
1620028 //
1620029 // Check if it is possible to get the new size:
1620030 // cannot be less
1620031 // then the original segment size.
1620032 //
1620033 if (proc_table[pid].domain_data == 0)
1620034 {
1620035     if (proc_table[pid].domain_text > requested_size)
1620036     {
1620037         requested_extra = 0;
1620038     }
1620039     else
1620040     {
1620041         requested_extra = requested_size
1620042             - proc_table[pid].domain_text;
1620043     }
1620044 }
1620045 else
1620046 {
1620047     if (proc_table[pid].domain_data > requested_size)
1620048     {
1620049         requested_extra = 0;
1620050     }
1620051     else
1620052     {
1620053         requested_extra = requested_size
1620054             - proc_table[pid].domain_data;
1620055     }
1620056 }
1620057 //
1620058 // Now make shure that the new value is a multiple
1620059 // of
1620060 // MEM_BLOCK_SIZE!
```



```
1620061 //
1620062 if (requested_extra % MEM_BLOCK_SIZE)
1620063 {
1620064     requested_extra =
1620065         (((requested_extra / MEM_BLOCK_SIZE) +
1620066            1) * MEM_BLOCK_SIZE);
1620067 }
1620068 //
1620069 // Now resize the process.
1620070 //
1620071 if (requested_extra == proc_table[pid].extra_data)
1620072 {
1620073     //
1620074     // Nothing have to change.
1620075     //
1620076     return (0);
1620077 }
1620078 else if (requested_extra < proc_table[pid].extra_data)
1620079 {
1620080     //
1620081     // Just reduce.
1620082     //
1620083     previous_extra = proc_table[pid].extra_data;
1620084     proc_table[pid].extra_data = requested_extra;
1620085     //
1620086     // Update process DATA segment inside the GDT
1620087     // table.
1620088     //
1620089     if (proc_table[pid].domain_data > 0)
1620090     {
1620091         gdt_segment (gdt_pid_to_segment_data (pid),
1620092                     (uint32_t)
1620093                     proc_table[pid].address_data,
1620094                     (uint32_t) ((proc_table
1620095                                 [pid].domain_data +
1620096                                 proc_table
1620097                                 [pid].extra_data) /
```

```
1620098                                     MEM_BLOCK_SIZE), 1, 0,
1620099                                     0);
1620100     }
1620101     else
1620102     {
1620103         gdt_segment (gdt_pid_to_segment_data (pid),
1620104                     (uint32_t)
1620105                     proc_table[pid].address_text,
1620106                     (uint32_t) ((proc_table
1620107                                 [pid].domain_text +
1620108                                 proc_table
1620109                                 [pid].extra_data) /
1620110                                 MEM_BLOCK_SIZE), 1, 0,
1620111                                     0);
1620112     }
1620113     //
1620114     // Release memory.
1620115     //
1620116     if (proc_table[pid].domain_data > 0)
1620117     {
1620118         status =
1620119             mb_reduce ((proc_table[pid].address_data +
1620120                       proc_table[pid].domain_data),
1620121                      proc_table[pid].extra_data,
1620122                      previous_extra);
1620123     }
1620124     else
1620125     {
1620126         status =
1620127             mb_reduce ((proc_table[pid].address_text +
1620128                       proc_table[pid].domain_text),
1620129                      proc_table[pid].extra_data,
1620130                      previous_extra);
1620131     }
1620132     //
1620133     if (status < 0)
1620134     {
```

```
1620135         //
1620136         // What happened?
1620137         //
1620138         k_perror (NULL);
1620139     }
1620140 }
1620141 else
1620142 {
1620143     //
1620144     // A bigger size was requested. Save previous
1620145     // information.
1620146     //
1620147     previous_address_text = proc_table[pid].address_text;
1620148     previous_domain_text = proc_table[pid].domain_text;
1620149     previous_address_data = proc_table[pid].address_data;
1620150     previous_domain_data = proc_table[pid].domain_data;
1620151     previous_extra = proc_table[pid].extra_data;
1620152     //
1620153     // Allocate memory for text,
1620154     // if text and data are inside the same address
1620155     // space;
1620156     // otherwise only the data segment is involved.
1620157     //
1620158     if (proc_table[pid].domain_data == 0)
1620159     {
1620160         allocated_text =
1620161             mb_alloc_size (proc_table[pid].domain_text +
1620162                 requested_extra);
1620163         //
1620164         if (allocated_text == 0)
1620165         {
1620166             errset (ENOMEM); // Not enough space.
1620167             return (-1);
1620168         }
1620169         //
1620170         if (DEBUG)
1620171         {
```

```
1620172         k_printf ("%s:%i:mb_alloc_size(%zi)",
1620173                 __FILE__, __LINE__,
1620174                 (proc_table[pid].domain_text
1620175                 + requested_extra));
1620176     }
1620177 }
1620178 //
1620179 // Allocate memory for data, if necessary.
1620180 //
1620181 if (proc_table[pid].domain_data > 0)
1620182 {
1620183     allocated_data =
1620184         mb_alloc_size (proc_table[pid].domain_data +
1620185                       requested_extra);
1620186     //
1620187     if (allocated_data == 0)
1620188     {
1620189         //
1620190         // Please note that, if we are here, no
1620191         // memory
1620192         // for the text was allocated!
1620193         //
1620194         errset (ENOMEM); // Not enough space.
1620195         return (-1);
1620196     }
1620197     //
1620198     if (DEBUG)
1620199     {
1620200         k_printf ("%s:%i:mb_alloc_size(%zi)",
1620201                 __FILE__, __LINE__,
1620202                 (proc_table[pid].domain_data
1620203                 + requested_extra));
1620204     }
1620205 }
1620206 //
1620207 // Copy the process text and, data in memory: if
1620208 // size is zero, no copy is made. But the text
```

```
1620209 // is
1620210 // copied only if text and data live together.
1620211 //
1620212 if (proc_table[pid].domain_data == 0)
1620213 {
1620214     memcpy ((void *) allocated_text,
1620215             (void *) proc_table[pid].address_text,
1620216             (size_t) (proc_table[pid].domain_text +
1620217                       proc_table[pid].extra_data));
1620218 }
1620219 else
1620220 {
1620221     memcpy ((void *) allocated_data,
1620222             (void *) proc_table[pid].address_data,
1620223             (size_t) (proc_table[pid].domain_data +
1620224                       proc_table[pid].extra_data));
1620225 }
1620226 //
1620227 // Update process information.
1620228 //
1620229 if (proc_table[pid].domain_data == 0)
1620230 {
1620231     proc_table[pid].address_text = allocated_text;
1620232 }
1620233 else
1620234 {
1620235     proc_table[pid].address_data = allocated_data;
1620236 }
1620237 proc_table[pid].extra_data = requested_extra;
1620238 //
1620239 // Update process TEXT segment inside the GDT
1620240 // table.
1620241 //
1620242 gdt_segment (gdt_pid_to_segment_text (pid),
1620243             (uint32_t) proc_table[pid].address_text,
1620244             (uint32_t) (proc_table[pid].domain_text /
1620245                       MEM_BLOCK_SIZE), 1, 1, 0);
```

```
1620246 //
1620247 // Update process DATA segment inside the GDT
1620248 // table.
1620249 //
1620250 if (proc_table[pid].domain_data > 0)
1620251 {
1620252     gdt_segment (gdt_pid_to_segment_data (pid),
1620253                 (uint32_t)
1620254                 proc_table[pid].address_data,
1620255                 (uint32_t) ((proc_table
1620256                             [pid].domain_data +
1620257                             proc_table
1620258                             [pid].extra_data) /
1620259                             MEM_BLOCK_SIZE), 1, 0,
1620260                 0);
1620261 }
1620262 else
1620263 {
1620264     gdt_segment (gdt_pid_to_segment_data (pid),
1620265                 (uint32_t)
1620266                 proc_table[pid].address_text,
1620267                 (uint32_t) ((proc_table
1620268                             [pid].domain_text +
1620269                             proc_table
1620270                             [pid].extra_data) /
1620271                             MEM_BLOCK_SIZE), 1, 0,
1620272                 0);
1620273 }
1620274 //
1620275 // Now release the old memory!
1620276 //
1620277 if (proc_table[pid].domain_data == 0)
1620278 {
1620279     mb_free (previous_address_text,
1620280             previous_domain_text + previous_extra);
1620281 }
1620282 else
```

```
1620283     {
1620284         mb_free (previous_address_data,
1620285                previous_domain_data + previous_extra);
1620286     }
1620287 }
1620288 //
1620289 // Ok.
1620290 //
1620291 return (0);
1620292 }
```

94.8.5 kernel/lib_s/s_chdir.c



Si veda la sezione [87.6](#).

```
1630001 #include <kernel/fs.h>
1630002 #include <errno.h>
1630003 #include <kernel/proc.h>
1630004 #include <kernel/lib_s.h>
1630005 //-----
1630006 int
1630007 s_chdir (pid_t pid, const char *path)
1630008 {
1630009     proc_t *ps;
1630010     inode_t *inode_directory;
1630011     int status;
1630012     char path_directory[PATH_MAX];
1630013     //
1630014     // Get process.
1630015     //
1630016     ps = proc_reference (pid);
1630017     //
1630018     // The full directory path is needed.
1630019     //
1630020     status = path_full (path, ps->path_cwd, path_directory);
1630021     if (status < 0)
1630022     {
```

```
1630023     return (-1);
1630024     }
1630025     //
1630026     // Try to load the new directory inode.
1630027     //
1630028     inode_directory = path_inode (pid, path_directory);
1630029     if (inode_directory == NULL)
1630030     {
1630031         //
1630032         // Cannot access the directory: it does not
1630033         // exists or
1630034         // permissions are not sufficient. Variable
1630035         // 'errno' is set by
1630036         // function 'inode_directory()'.
1630037         //
1630038         errset (errno);
1630039         return (-1);
1630040     }
1630041     //
1630042     // Inode loaded: release the old directory and set
1630043     // the new one.
1630044     //
1630045     inode_put (ps->inode_cwd);
1630046     //
1630047     ps->inode_cwd = inode_directory;
1630048     strncpy (ps->path_cwd, path_directory, PATH_MAX);
1630049     //
1630050     // Return.
1630051     //
1630052     return (0);
1630053 }
```


94.8.6 kernel/lib_s/s_chmod.c



Si veda la sezione [87.7](#).

```
1640001 #include <kernel/fs.h>
1640002 #include <errno.h>
1640003 #include <kernel/proc.h>
1640004 #include <kernel/lib_s.h>
1640005 //-----
1640006 int
1640007 s_chmod (pid_t pid, const char *path, mode_t mode)
1640008 {
1640009     proc_t *ps;
1640010     inode_t *inode;
1640011     //
1640012     // Get process.
1640013     //
1640014     ps = proc_reference (pid);
1640015     //
1640016     // Try to load the file inode.
1640017     //
1640018     inode = path_inode (pid, path);
1640019     if (inode == NULL)
1640020     {
1640021         //
1640022         // Cannot access the file: it does not exists or
1640023         // permissions are
1640024         // not sufficient. Variable 'errno' is set by
1640025         // function
1640026         // 'inode_directory()'.
1640027         //
1640028         return (-1);
1640029     }
1640030     //
1640031     // Verify to be root or to be the owner.
1640032     //
1640033     if (ps->euid != 0 && ps->euid != inode->uid)
1640034     {
```

```
1640035     errset (EACCES); // Permission denied.
1640036     return (-1);
1640037 }
1640038 //
1640039 // Update the mode: the file type is kept and the
1640040 // rest is taken form the parameter 'mode'.
1640041 //
1640042 inode->mode = (S_IFMT & inode->mode) | (~S_IFMT & mode);
1640043 //
1640044 // Save and release the inode.
1640045 //
1640046 inode->changed = 1;
1640047 inode_save (inode);
1640048 inode_put (inode);
1640049 //
1640050 // Return.
1640051 //
1640052 return (0);
1640053 }
```

94.8.7 kernel/lib_s/s_chown.c

<<

Si veda la sezione [87.8](#).

```
1650001 #include <kernel/fs.h>
1650002 #include <errno.h>
1650003 #include <kernel/proc.h>
1650004 #include <kernel/lib_s.h>
1650005 //-----
1650006 int
1650007 s_chown (pid_t pid, const char *path, uid_t uid, gid_t gid)
1650008 {
1650009     proc_t *ps;
1650010     inode_t *inode;
1650011 //
1650012 // Get process.
1650013 //
```

```
1650014 ps = proc_reference (pid);
1650015 //
1650016 // Must be root, as the ability to change group is
1650017 // not considered.
1650018 //
1650019 if (ps->euid != 0)
1650020 {
1650021     errset (EPERM); // Operation not permitted.
1650022     return (-1);
1650023 }
1650024 //
1650025 // Try to load the file inode.
1650026 //
1650027 inode = path_inode (pid, path);
1650028 if (inode == NULL)
1650029 {
1650030     //
1650031     // Cannot access the file: it does not exists or
1650032     // permissions are
1650033     // not sufficient. Variable 'errno' is set by
1650034     // function
1650035     // 'inode_directory()'.
1650036     //
1650037     return (-1);
1650038 }
1650039 //
1650040 // Update the owner and group.
1650041 //
1650042 if (uid != -1)
1650043 {
1650044     inode->uid = uid;
1650045     inode->changed = 1;
1650046 }
1650047 if (gid != -1)
1650048 {
1650049     inode->gid = gid;
1650050     inode->changed = 1;
```

```
1650051     }
1650052     //
1650053     // Save and release the inode.
1650054     //
1650055     inode_save (inode);
1650056     inode_put (inode);
1650057     //
1650058     // Return.
1650059     //
1650060     return (0);
1650061 }
```

94.8.8 kernel/lib_s/s_clock.c

<<

Si veda la sezione [87.9](#).

```
1660001 #include <kernel/lib_s.h>
1660002 //-----
1660003 extern clock_t _clock_kernel; // uint64_t
1660004 //-----
1660005 clock_t
1660006 s_clock (pid_t pid)
1660007 {
1660008     return (_clock_kernel);
1660009 }
```

94.8.9 kernel/lib_s/s_close.c

<<

Si veda la sezione [87.10](#).

```
1670001 #include <kernel/proc.h>
1670002 #include <fcntl.h>
1670003 #include <errno.h>
1670004 //-----
1670005 int
1670006 s_close (pid_t pid, int fdn)
1670007 {
```

```
1670008     fd_t *fd;
1670009     int status;
1670010     //
1670011     // Get file descriptor.
1670012     //
1670013     fd = fd_reference (pid, &fdn);
1670014     if (fd == NULL || fd->file == NULL
1670015         || (fd->file->inode == NULL
1670016             && fd->file->sock == NULL))
1670017     {
1670018         errset (EBADF);    // Bad file descriptor.
1670019         return (-1);
1670020     }
1670021     //
1670022     //
1670023     //
1670024     if (fd->file->inode != NULL)    // Inode
1670025     {
1670026         //
1670027         // File descriptor with inode.
1670028         //
1670029         // If it is a pipe, some special things must be
1670030         // done.
1670031         //
1670032         if (S_ISFIFO (fd->file->inode->mode))
1670033         {
1670034             if (fd->fl_flags & O_RDONLY)
1670035             {
1670036                 fd->file->inode->pipe_ref_read--;
1670037                 if (fd->file->inode->pipe_ref_read == 0)
1670038                 {
1670039                     proc_wakeup_pipe_write (fd->file->inode);
1670040                 }
1670041             }
1670042             //
1670043             if (fd->fl_flags & O_WRONLY)
1670044             {
```

```
1670045         fd->file->inode->pipe_ref_write--;
1670046         if (fd->file->inode->pipe_ref_write == 0)
1670047             {
1670048                 proc_wakeup_pipe_read (fd->file->inode);
1670049             }
1670050     }
1670051 }
1670052 }
1670053 else // Socket
1670054 {
1670055     //
1670056     // File descriptor with socket.
1670057     //
1670058     status = tcp_close (fd->file->sock);
1670059     if (status < 0
1670060         && (errno == EINPROGRESS || errno == EALREADY))
1670061     {
1670062         errset (errno);
1670063         return (status);
1670064     }
1670065     //
1670066     // Otherwise, the socket is closed and can
1670067     // continue
1670068     // with the other references.
1670069     //
1670070 }
1670071 //
1670072 // Reduce references inside the file table item
1670073 // and remove item if it reaches zero.
1670074 //
1670075 fd->file->references--;
1670076 if (fd->file->references == 0)
1670077     {
1670078         fd->file->oflags = 0;
1670079         fd->file->inode = NULL;
1670080         //
1670081         // Put inode, or release the socket, because
```

```

1670082         // there are no more
1670083         // file references.
1670084         //
1670085         if (fd->file->inode != NULL)
1670086             {
1670087                 inode_put (fd->file->inode);
1670088             }
1670089         if (fd->file->sock != NULL)
1670090             {
1670091                 sock_put (fd->file->sock);
1670092             }
1670093     }
1670094     //
1670095     // Remove file descriptor.
1670096     //
1670097     fd->fl_flags = 0;
1670098     fd->fd_flags = 0;
1670099     fd->file = NULL;
1670100     //
1670101     //
1670102     //
1670103     return (0);
1670104 }

```

94.8.10 kernel/lib_s/s_connect.c

Si veda la sezione [87.11](#).

```

1680001 #include <kernel/proc.h>
1680002 #include <kernel/lib_s.h>
1680003 #include <kernel/lib_k.h>
1680004 #include <kernel/net/route.h>
1680005 #include <errno.h>
1680006 #include <fcntl.h>
1680007 #include <sys/socket.h>
1680008 #include <arpa/inet.h>
1680009 //-----

```

```
1680010 int
1680011 s_connect (pid_t pid, int sfdn,
1680012           const struct sockaddr *addr, socklen_t addrlen)
1680013 {
1680014     fd_t *sfd;
1680015     struct sockaddr_in *sin;
1680016     clock_t clock_time;
1680017     int i;
1680018     int status;
1680019     //
1680020     // Get file descriptor and verify that it is a
1680021     // socket.
1680022     //
1680023     sfd = fd_reference (pid, &sfdn);
1680024     if (sfd == NULL || sfd->file == NULL)
1680025     {
1680026         errset (EBADF);    // Bad file descriptor.
1680027         return (-1);
1680028     }
1680029     if (sfd->file->sock == NULL)
1680030     {
1680031         errset (ENOTSOCK);    // Not a socket.
1680032         return (-1);
1680033     }
1680034     //
1680035     // Verify to have a valid address pointer.
1680036     //
1680037     if (addr == NULL)
1680038     {
1680039         errset (EINVAL);
1680040         return (-1);
1680041     }
1680042     //
1680043     // Check minimal address size.
1680044     //
1680045     if (addrlen < sizeof (struct sockaddr))
1680046     {
```



```
1680047     errset (EINVAL);
1680048     return (-1);
1680049 }
1680050 //
1680051 //
1680052 //
1680053 if (addr->sa_family == AF_INET)
1680054 {
1680055     sin = (struct sockaddr_in *) addr;
1680056     //
1680057     // Check to have the destination IP address.
1680058     //
1680059     if (sin->sin_addr.s_addr == 0)
1680060     {
1680061         //
1680062         // This is not valid.
1680063         //
1680064         errset (EADDRNOTAVAIL);
1680065         return (-1);
1680066     }
1680067     //
1680068     // If it is a TCP/UDP protocol, must have valid
1680069     // ports.
1680070     //
1680071     if (sfd->file->sock->protocol == IPPROTO_TCP
1680072         || sfd->file->sock->protocol == IPPROTO_UDP)
1680073     {
1680074         //
1680075         // Remote port.
1680076         //
1680077         if (sin->sin_port == 0)
1680078         {
1680079             //
1680080             // Missing the remote port.
1680081             //
1680082             errset (EADDRNOTAVAIL);
1680083             return (-1);
```

```
1680084     }
1680085     //
1680086     // Local port.
1680087     //
1680088     if (sfd->file->sock->lport == 0)
1680089     {
1680090         //
1680091         // Must find a free one.
1680092         //
1680093         sfd->file->sock->lport = sock_free_port ();
1680094         if (sfd->file->sock->lport == 0)
1680095         {
1680096             //
1680097             // No port is available.
1680098             //
1680099             errset (EAGAIN);
1680100             return (-1);
1680101         }
1680102     }
1680103 }
1680104 //
1680105 // Update the socket, but not a TCP connection
1680106 // already
1680107 // working (TCP not connected has a zeroed
1680108 // 'tcp.conn' filled).
1680109 //
1680110 if (sfd->file->sock->tcp.conn == 0
1680111     || sfd->file->sock->tcp.conn == TCP_CLOSE)
1680112 {
1680113     //
1680114     // Update the socket.
1680115     //
1680116     sfd->file->sock->family = sin->sin_family;
1680117     sfd->file->sock->raddr =
1680118         ntohl (sin->sin_addr.s_addr);
1680119     sfd->file->sock->rport = ntohs (sin->sin_port);
1680120     //
```

```
1680121         // Reset read packets clock time.
1680122         //
1680123         clock_time = s_clock (pid);
1680124         for (i = 0; i < IP_MAX_PACKETS; i++)
1680125             {
1680126                 sfd->file->sock->read.clock[i] = clock_time;
1680127             }
1680128     }
1680129     else
1680130     {
1680131         //
1680132         // It *is* a TCP connection already working:
1680133         // verify that
1680134         // the socket is set as expected
1680135         //
1680136         if (sfd->file->sock->family !=
1680137             sin->sin_family
1680138             || sfd->file->sock->raddr !=
1680139             ntohl (sin->sin_addr.s_addr)
1680140             || sfd->file->sock->rport !=
1680141             ntohs (sin->sin_port))
1680142             {
1680143                 //
1680144                 // The socket address is changed!
1680145                 //
1680146                 errset (EISCONN);
1680147                 return (-1);
1680148             }
1680149     }
1680150     //
1680151     // If it is a TCP, not already working, should
1680152     // connect.
1680153     // The function 'tcp_connect()' will verify the
1680154     // connection
1680155     // status.
1680156     //
1680157     if (sfd->file->sock->protocol == IPPROTO_TCP)
```

```
1680158     {
1680159         //
1680160         // Be shure to have a source address.
1680161         //
1680162         if (sfd->file->sock->laddr == 0)
1680163         {
1680164             //
1680165             // Default source address: get the
1680166             // source address from the
1680167             // routing table, based on the
1680168             // destination.
1680169             //
1680170             sfd->file->sock->laddr
1680171             =
1680172             route_remote_to_local (sfd->file->
1680173                                   sock->raddr);
1680174             if (sfd->file->sock->laddr ==
1680175                 ((h_addr_t) - 1))
1680176             {
1680177                 errset (errno);
1680178                 return (-1);
1680179             }
1680180         }
1680181         //
1680182         // Call tcp_connect ().
1680183         //
1680184         status = tcp_connect (sfd->file->sock);
1680185         if (status)
1680186         {
1680187             errset (errno);
1680188         }
1680189         return (status);
1680190     }
1680191 }
1680192 else
1680193 {
1680194     //
```

```
1680195 // It is not AF_INET: unsupported address
1680196 // family.
1680197 //
1680198     errset (EAFNOSUPPORT);
1680199     return (-1);
1680200 }
1680201 //
1680202 // Ok.
1680203 //
1680204     return (0);
1680205 }
```

94.8.11 kernel/lib_s/s_dup.c

Si veda la sezione [87.12](#).

```
1690001 #include <kernel/lib_s.h>
1690002 #include <kernel/fs.h>
1690003 //-----
1690004 int
1690005 s_dup (pid_t pid, int fdn_old)
1690006 {
1690007     return (fd_dup (pid, fdn_old, 0));
1690008 }
```

94.8.12 kernel/lib_s/s_dup2.c

Si veda la sezione [87.12](#).

```
1700001 #include <kernel/proc.h>
1700002 #include <kernel/lib_s.h>
1700003 #include <errno.h>
1700004 #include <fcntl.h>
1700005 //-----
1700006 int
1700007 s_dup2 (pid_t pid, int fdn_old, int fdn_new)
1700008 {
```

```
170009  proc_t *ps;
170010  int status;
170011  //
170012  // Get process.
170013  //
170014  ps = proc_reference (pid);
170015  //
170016  // Verify if 'fdn_old' is a valid value.
170017  //
170018  if (fdn_old < 0 ||
170019      fdn_old >= OPEN_MAX || ps->fd[fdn_old].file == NULL)
170020  {
170021      errset (EBADF);    // Bad file descriptor.
170022      return (-1);
170023  }
170024  //
170025  // Check if 'fd_old' and 'fd_new' are the same.
170026  //
170027  if (fdn_old == fdn_new)
170028  {
170029      return (fdn_new);
170030  }
170031  //
170032  // Close 'fdn_new' if it is open and copy 'fdn_old'
170033  // into it.
170034  //
170035  if (ps->fd[fdn_new].file != NULL)
170036  {
170037      status = s_close (pid, fdn_new);
170038      if (status != 0)
170039      {
170040          return (-1);
170041      }
170042  }
170043  ps->fd[fdn_new].fl_flags = ps->fd[fdn_old].fl_flags;
170044  ps->fd[fdn_new].fd_flags =
170045  ps->fd[fdn_old].fd_flags & ~FD_CLOEXEC;
```

```
1700046     ps->fd[fdn_new].file = ps->fd[fdn_old].file;
1700047     ps->fd[fdn_new].file->references++;
1700048     return (fdn_new);
1700049 }
```

94.8.13 kernel/lib_s/s_fchmod.c

Si veda la sezione [87.7](#).

```
1710001 #include <kernel/proc.h>
1710002 #include <kernel/lib_s.h>
1710003 #include <sys/stat.h>
1710004 #include <errno.h>
1710005 //-----
1710006 int
1710007 s_fchmod (pid_t pid, int fdn, mode_t mode)
1710008 {
1710009     proc_t *ps;
1710010     inode_t *inode;
1710011     //
1710012     // Get process.
1710013     //
1710014     ps = proc_reference (pid);
1710015     //
1710016     // Verify if the file descriptor is valid.
1710017     //
1710018     if (ps->fd[fdn].file == NULL)
1710019     {
1710020         errset (EBADF);    // Bad file descriptor.
1710021         return (-1);
1710022     }
1710023     //
1710024     // Reach the inode.
1710025     //
1710026     inode = ps->fd[fdn].file->inode;
1710027     //
1710028     // If the Inode does not exist, exit with error.
```

```
1710029 //
1710030 if (inode == NULL)
1710031 {
1710032     errset (ENOENT);
1710033     return (-1);
1710034 }
1710035 //
1710036 // Verify to be the owner, or at least to be UID ==
1710037 // 0.
1710038 //
1710039 if (ps->euid != inode->uid && ps->euid != 0)
1710040 {
1710041     errset (EACCES); // Permission denied.
1710042     return (-1);
1710043 }
1710044 //
1710045 // Update the mode: the file type is kept and the
1710046 // rest is taken form the parameter 'mode'.
1710047 //
1710048 inode->mode = (S_IFMT & inode->mode) | (~S_IFMT & mode);
1710049 //
1710050 // Save the inode.
1710051 //
1710052 inode->changed = 1;
1710053 inode_save (inode);
1710054 //
1710055 // Return.
1710056 //
1710057 return (0);
1710058 }
```

94.8.14 kernel/lib_s/s_fchown.c



Si veda la sezione [87.8](#).

```
1720001 #include <kernel/proc.h>
1720002 #include <kernel/lib_s.h>
```



```
1720003 #include <errno.h>
1720004 //-----
1720005 int
1720006 s_fchown (pid_t pid, int fdn, uid_t uid, gid_t gid)
1720007 {
1720008     proc_t *ps;
1720009     inode_t *inode;
1720010     //
1720011     // Get process.
1720012     //
1720013     ps = proc_reference (pid);
1720014     //
1720015     // Verify if the file descriptor is valid.
1720016     //
1720017     if (ps->fd[fdn].file == NULL)
1720018     {
1720019         errset (EBADF);    // Bad file descriptor.
1720020         return (-1);
1720021     }
1720022     //
1720023     // Reach the inode.
1720024     //
1720025     inode = ps->fd[fdn].file->inode;
1720026     //
1720027     // If the Inode does not exist, exit with error.
1720028     //
1720029     if (inode == NULL)
1720030     {
1720031         errset (ENOENT);
1720032         return (-1);
1720033     }
1720034     //
1720035     // Verify to be root, as the ability to change group
1720036     // is not taken into consideration.
1720037     //
1720038     if (ps->euid != 0)
1720039     {
```

```
1720040     errset (EACCES); // Permission denied.
1720041     return (-1);
1720042 }
1720043 //
1720044 // Update the ownership.
1720045 //
1720046 if (uid != -1)
1720047 {
1720048     inode->uid = uid;
1720049     inode->changed = 1;
1720050 }
1720051 if (gid != -1)
1720052 {
1720053     inode->gid = gid;
1720054     inode->changed = 1;
1720055 }
1720056 //
1720057 // Save the inode.
1720058 //
1720059 inode->changed = 1;
1720060 inode_save (inode);
1720061 //
1720062 // Return.
1720063 //
1720064 return (0);
1720065 }
```

94.8.15 kernel/lib_s/s_fcntl.c



Si veda la sezione [87.18](#).

```
1730001 #include <kernel/proc.h>
1730002 #include <kernel/lib_s.h>
1730003 #include <kernel/fs.h>
1730004 #include <errno.h>
1730005 #include <fcntl.h>
1730006 //-----
```

```
1730007 int
1730008 s_fcntl (pid_t pid, int fdn, int cmd, int arg)
1730009 {
1730010     proc_t *ps;
1730011     int mask;
1730012     //
1730013     // Get process.
1730014     //
1730015     ps = proc_reference (pid);
1730016     //
1730017     // Verify if the file descriptor is valid.
1730018     //
1730019     if (ps->fd[fdn].file == NULL)
1730020     {
1730021         errset (EBADF);    // Bad file descriptor.
1730022         return (-1);
1730023     }
1730024     //
1730025     //
1730026     //
1730027     switch (cmd)
1730028     {
1730029         case F_DUPFD:
1730030             return (fd_dup (pid, fdn, arg));
1730031             break;
1730032         case F_GETFD:
1730033             return (ps->fd[fdn].fd_flags);
1730034             break;
1730035         case F_SETFD:
1730036             ps->fd[fdn].fd_flags = arg;
1730037             return (0);
1730038         case F_GETFL:
1730039             return (ps->fd[fdn].fl_flags);
1730040         case F_SETFL:
1730041             //
1730042             // Calculate a mask with bits that are *not* to
1730043             // be set.
```

```

1730044 //
1730045 mask =
1730046     (O_ACCMODE | O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);
1730047 //
1730048 // Set to zero the bits that are not to be set
1730049 // from
1730050 // the argument.
1730051 //
1730052 arg = (arg & ~mask);
1730053 //
1730054 // Set to zero the bit that *are* to be set.
1730055 //
1730056 ps->fd[fdn].fl_flags &= mask;
1730057 //
1730058 // Set the bits, already filtered inside the
1730059 // argument.
1730060 //
1730061 ps->fd[fdn].fl_flags |= arg;
1730062 //
1730063 return (0);
1730064 default:
1730065     errset (EINVAL); // Not implemented.
1730066     return (-1);
1730067 }
1730068 }

```

94.8.16 kernel/lib_s/s_fork.c



Si veda la sezione [87.19](#).

```

1740001 #include <kernel/proc.h>
1740002 #include <errno.h>
1740003 #include <fcntl.h>
1740004 #include <kernel/lib_k.h>
1740005 #include <kernel/lib_s.h>
1740006 //-----
1740007 #define DEBUG 0

```

```
1740008 //-----
1740009 extern uint32_t proc_stack_pointer;
1740010 //-----
1740011 pid_t
1740012 s_fork (pid_t ppid)
1740013 {
1740014     pid_t pid;
1740015     pid_t zombie;
1740016     addr_t allocated_text = 0;
1740017     addr_t allocated_data = 0;
1740018     addr_t addr_stack_pointer = 0;
1740019     int fdn;
1740020     uint16_t segment_descriptor;
1740021     int sig;
1740022     //
1740023     // Find a free PID.
1740024     //
1740025     for (pid = 1; pid < PROCESS_MAX; pid++)
1740026     {
1740027         if (proc_table[pid].status == PROC_EMPTY)
1740028         {
1740029             break;
1740030         }
1740031     }
1740032     if (pid >= PROCESS_MAX)
1740033     {
1740034         //
1740035         // There is no free pid.
1740036         //
1740037         errset (ENOMEM); // Not enough space.
1740038         return (-1);
1740039     }
1740040     //
1740041     // Before allocating a new process, must check if
1740042     // there are some
1740043     // zombie slots, still with original segment data:
1740044     // should reset
```

```
1740045 // them now!
1740046 //
1740047 for (zombie = 1; zombie < PROCESS_MAX; zombie++)
1740048 {
1740049     if (proc_table[zombie].status == PROC_ZOMBIE
1740050         && (proc_table[zombie].address_text != 0
1740051             || proc_table[zombie].domain_text != 0))
1740052     {
1740053         proc_table[zombie].address_text = (addr_t) 0;
1740054         proc_table[zombie].domain_text = (size_t) 0;
1740055         proc_table[zombie].address_data = (addr_t) 0;
1740056         proc_table[zombie].domain_data = (size_t) 0;
1740057         proc_table[zombie].domain_stack = (size_t) 0;
1740058         proc_table[zombie].extra_data = (size_t) 0;
1740059         proc_table[zombie].sp = 0;
1740060     }
1740061 }
1740062 //
1740063 // Allocate memory for text, if text and data are
1740064 // inside
1740065 // the same address space.
1740066 //
1740067 if (proc_table[ppid].domain_data == 0)
1740068 {
1740069     allocated_text =
1740070         mb_alloc_size (proc_table[ppid].domain_text +
1740071                       proc_table[ppid].extra_data);
1740072     //
1740073     if (allocated_text == 0)
1740074     {
1740075         errset (ENOMEM); // Not enough space.
1740076         return ((pid_t) - 1);
1740077     }
1740078     //
1740079     if (DEBUG)
1740080     {
1740081         k_printf ("%s:%i:mb_alloc_size(%zi)",
```

```
1740082         __FILE__, __LINE__,
1740083         (proc_table[ppid].domain_text
1740084         + proc_table[ppid].extra_data));
1740085     }
1740086 }
1740087 //
1740088 // Allocate memory for data, if necessary.
1740089 //
1740090 if (proc_table[ppid].domain_data > 0)
1740091 {
1740092     allocated_data =
1740093         mb_alloc_size (proc_table[ppid].domain_data +
1740094         proc_table[ppid].extra_data);
1740095     //
1740096     if (allocated_data == 0)
1740097     {
1740098         //
1740099         // Please note that, if we are here, no
1740100         // memory
1740101         // for the text was allocated!
1740102         //
1740103         errset (ENOMEM);        // Not enough space.
1740104         return ((pid_t) - 1);
1740105     }
1740106     //
1740107     if (DEBUG)
1740108     {
1740109         k_printf ("%s:%i:mb_alloc_size(%zi)",
1740110         __FILE__, __LINE__,
1740111         (proc_table[ppid].domain_data
1740112         + proc_table[ppid].extra_data));
1740113     }
1740114 }
1740115 //
1740116 // Copy the process text and, data in memory: if
1740117 // size is zero, no copy is made. But the text is
1740118 // copied only if text and data live together.
```

```
1740119 //
1740120 if (proc_table[ppid].domain_data == 0)
1740121 {
1740122     memcpy ((void *) allocated_text,
1740123            (void *) proc_table[ppid].address_text,
1740124            (size_t) (proc_table[ppid].domain_text
1740125                    + proc_table[ppid].extra_data));
1740126 }
1740127 else
1740128 {
1740129     memcpy ((void *) allocated_data,
1740130            (void *) proc_table[ppid].address_data,
1740131            (size_t) (proc_table[ppid].domain_data
1740132                    + proc_table[ppid].extra_data));
1740133 }
1740134 //
1740135 // Allocate the new PID inside the 'proc_table[]'.
1740136 //
1740137 proc_table[pid].ppid = ppid;
1740138 proc_table[pid].pgrp = proc_table[ppid].pgrp;
1740139 proc_table[pid].uid = proc_table[ppid].uid;
1740140 proc_table[pid].euid = proc_table[ppid].euid;
1740141 proc_table[pid].suid = proc_table[ppid].suid;
1740142 proc_table[pid].gid = proc_table[ppid].gid;
1740143 proc_table[pid].egid = proc_table[ppid].egid;
1740144 proc_table[pid].sgid = proc_table[ppid].sgid;
1740145 proc_table[pid].device_tty = proc_table[ppid].device_tty;
1740146 proc_table[pid].sig_status = 0;
1740147 proc_table[pid].sig_ignore = 0;
1740148 //
1740149 for (sig = 0; sig < MAX_SIGNALS; sig++)
1740150 {
1740151     proc_table[pid].sig_handler[sig]
1740152     = proc_table[ppid].sig_handler[sig];
1740153 }
1740154 //
1740155 proc_table[pid].usage = 0;
```



```
1740156     proc_table[pid].status = PROC_CREATED;
1740157     proc_table[pid].wakeup_events = 0;
1740158     proc_table[pid].wakeup_signal = 0;
1740159     proc_table[pid].wakeup_timer = 0;
1740160     //
1740161     if (proc_table[ppid].domain_data != 0)
1740162     {
1740163         proc_table[pid].address_text =
1740164             proc_table[ppid].address_text;
1740165     }
1740166     else
1740167     {
1740168         proc_table[pid].address_text = allocated_text;
1740169     }
1740170     proc_table[pid].domain_text =
1740171         proc_table[ppid].domain_text;
1740172     proc_table[pid].address_data = allocated_data;
1740173     proc_table[pid].domain_data =
1740174         proc_table[ppid].domain_data;
1740175     proc_table[pid].domain_stack =
1740176         proc_table[ppid].domain_stack;
1740177     proc_table[pid].extra_data = proc_table[ppid].extra_data;
1740178     proc_table[pid].sp = proc_stack_pointer;
1740179     proc_table[pid].ret = 0;
1740180     proc_table[pid].inode_cwd = proc_table[ppid].inode_cwd;
1740181     proc_table[pid].umask = proc_table[ppid].umask;
1740182     strncpy (proc_table[pid].name, proc_table[ppid].name,
1740183             PATH_MAX);
1740184     strncpy (proc_table[pid].path_cwd,
1740185             proc_table[ppid].path_cwd, PATH_MAX);
1740186     //
1740187     // Update process TEXT segment inside the GDT table.
1740188     //
1740189     gdt_segment (gdt_pid_to_segment_text (pid),
1740190                 (uint32_t) proc_table[pid].address_text,
1740191                 (uint32_t) (proc_table[pid].domain_text /
1740192                             4096), 1, 1, 0);
```

```
1740193 //
1740194 // Update process DATA segment inside the GDT table.
1740195 //
1740196 if (proc_table[pid].domain_data > 0)
1740197 {
1740198     gdt_segment (gdt_pid_to_segment_data (pid),
1740199                 (uint32_t) proc_table[pid].address_data,
1740200                 (uint32_t) ((proc_table[pid].domain_data
1740201                             +
1740202                             proc_table[pid].extra_data)
1740203                             / 4096), 1, 0, 0);
1740204 }
1740205 else
1740206 {
1740207     gdt_segment (gdt_pid_to_segment_data (pid),
1740208                 (uint32_t) proc_table[pid].address_text,
1740209                 (uint32_t) ((proc_table[pid].domain_text
1740210                             +
1740211                             proc_table[pid].extra_data)
1740212                             / 4096), 1, 0, 0);
1740213 }
1740214 //
1740215 // -----
1740216 // Might reload the GDT table, but it is not
1740217 // necessarily.
1740218 // Anyway, if you do it, nothing change. :-)
1740219 //
1740220 // gdt_load (&gdt_register);
1740221 // -----
1740222 //
1740223 // Increase inode references for the working
1740224 // directory.
1740225 //
1740226 proc_table[pid].inode_cwd->references++;
1740227 //
1740228 // Duplicate valid file descriptors.
1740229 //
```

```
1740230     for (fdn = 0; fdn < OPEN_MAX; fdn++)
1740231     {
1740232         if (proc_table[ppid].fd[fdn].file != NULL
1740233             && (proc_table[ppid].fd[fdn].file->inode !=
1740234                 NULL
1740235                 || proc_table[ppid].fd[fdn].file->sock !=
1740236                 NULL))
1740237         {
1740238             //
1740239             // Copy to the forked process.
1740240             //
1740241             proc_table[pid].fd[fdn].fl_flags
1740242                 = proc_table[ppid].fd[fdn].fl_flags;
1740243             proc_table[pid].fd[fdn].fd_flags
1740244                 = proc_table[ppid].fd[fdn].fd_flags;
1740245             proc_table[pid].fd[fdn].file =
1740246                 proc_table[ppid].fd[fdn].file;
1740247             //
1740248             // Increment file reference.
1740249             //
1740250             proc_table[ppid].fd[fdn].file->references++;
1740251             //
1740252             // Check if it is a pipe and increment
1740253             // specific
1740254             // read/write reference counters inside the
1740255             // inode.
1740256             //
1740257             if (proc_table[ppid].fd[fdn].file->inode !=
1740258                 NULL
1740259                 && S_ISFIFO (proc_table[ppid].fd[fdn].
1740260                             file->inode->mode))
1740261             {
1740262                 if (proc_table[ppid].fd[fdn].
1740263                     fl_flags & O_RDONLY)
1740264                 {
1740265                     proc_table[ppid].fd[fdn].file->
1740266                         inode->pipe_ref_read++;
```

```
1740267     }
1740268     //
1740269     if (proc_table[ppid].fd[fdn].
1740270         fl_flags & O_WRONLY)
1740271     {
1740272         proc_table[ppid].fd[fdn].file->
1740273             inode->pipe_ref_write++;
1740274     }
1740275 }
1740276 }
1740277 }
1740278 //
1740279 // Change segment descriptor values inside the
1740280 // stack,
1740281 // for: DS==ES==FS==GS.
1740282 //
1740283 // First calculate the absolute stack section
1740284 // address, from the
1740285 // kernel point of view.
1740286 //
1740287 if (allocated_data > 0)
1740288 {
1740289     addr_stack_pointer = allocated_data;
1740290 }
1740291 else
1740292 {
1740293     addr_stack_pointer = allocated_text;
1740294 }
1740295 //
1740296 // Then calculate the effective new stack pointer.
1740297 //
1740298 addr_stack_pointer += proc_table[pid].sp;
1740299 //
1740300 // Then calculate the segment descriptor, to be
1740301 // written
1740302 // inside the new process stack.
1740303 //
```

```
1740304     segment_descriptor = (gdt_pid_to_segment_data (pid) * 8);
1740305     //
1740306     // Then copy inside the stack the new values for
1740307     // data segments.
1740308     //
1740309     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1740310            (addr_stack_pointer + 28),
1740311            &segment_descriptor,
1740312            (sizeof segment_descriptor), NULL);
1740313     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1740314            (addr_stack_pointer + 32),
1740315            &segment_descriptor,
1740316            (sizeof segment_descriptor), NULL);
1740317     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1740318            (addr_stack_pointer + 36),
1740319            &segment_descriptor,
1740320            (sizeof segment_descriptor), NULL);
1740321     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1740322            (addr_stack_pointer + 40),
1740323            &segment_descriptor,
1740324            (sizeof segment_descriptor), NULL);
1740325     //
1740326     // Change segment descriptor value inside the stack
1740327     // for CS,
1740328     // if so is necessary.
1740329     //
1740330     segment_descriptor = (gdt_pid_to_segment_text (pid) * 8);
1740331     //
1740332     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1740333            (addr_stack_pointer + 48),
1740334            &segment_descriptor,
1740335            (sizeof segment_descriptor), NULL);
1740336     //
1740337     // Set it ready.
1740338     //
1740339     proc_table[pid].status = PROC_READY;
1740340     //
```

```
1740341 // Return the new PID.
1740342 //
1740343 return (pid);
1740344 }
```

94.8.17 kernel/lib_s/s_fstat.c

<<

Si veda la sezione [87.55](#).

```
1750001 #include <kernel/proc.h>
1750002 #include <kernel/lib_s.h>
1750003 #include <errno.h>
1750004 #include <fcntl.h>
1750005 //-----
1750006 int
1750007 s_fstat (pid_t pid, int fdn, struct stat *buffer)
1750008 {
1750009     proc_t *ps;
1750010     inode_t *inode;
1750011     //
1750012     // Get process.
1750013     //
1750014     ps = proc_reference (pid);
1750015     //
1750016     // Verify if the file descriptor is valid.
1750017     //
1750018     if (ps->fd[fdn].file == NULL)
1750019     {
1750020         errset (EBADF); // Bad file descriptor.
1750021         return (-1);
1750022     }
1750023     //
1750024     // Reach the inode.
1750025     //
1750026     inode = ps->fd[fdn].file->inode;
1750027     //
1750028     // If the Inode does not exist, exit with error.
```

```
1750029 //
1750030 if (inode == NULL)
1750031 {
1750032     errset (ENOENT);
1750033     return (-1);
1750034 }
1750035 //
1750036 // Inode loaded: update the buffer.
1750037 //
1750038 buffer->st_dev = inode->sb->device;
1750039 buffer->st_ino = inode->ino;
1750040 buffer->st_mode = inode->mode;
1750041 buffer->st_nlink = inode->links;
1750042 buffer->st_uid = inode->uid;
1750043 buffer->st_gid = inode->gid;
1750044 if (S_ISBLK (buffer->st_mode)
1750045     || S_ISCHR (buffer->st_mode))
1750046 {
1750047     buffer->st_rdev = inode->direct[0];
1750048 }
1750049 else
1750050 {
1750051     buffer->st_rdev = 0;
1750052 }
1750053 buffer->st_size = inode->size;
1750054 buffer->st_atime = inode->time;           // All times
1750055 // are the
1750056 // same for
1750057 buffer->st_mtime = inode->time;         // Minix 1
1750058 // file
1750059 // system.
1750060 buffer->st_ctime = inode->time;         //
1750061 buffer->st_blksize = inode->sb->blksize;
1750062 buffer->st_blocks = inode->blkcnt;
1750063 //
1750064 // If the inode is a device special file, the
1750065 // 'st_rdev' value is
```

```
1750066 // taken from the first direct zone (as of Minix 1
1750067 // organization).
1750068 //
1750069 if (S_ISBLK (inode->mode) || S_ISCHR (inode->mode))
1750070 {
1750071     buffer->st_rdev = inode->direct[0];
1750072 }
1750073 else
1750074 {
1750075     buffer->st_rdev = 0;
1750076 }
1750077 //
1750078 // Return.
1750079 //
1750080 return (0);
1750081 }
```

94.8.18 kernel/lib_s/s_ipconfig.c

<<

Si veda la sezione [87.28](#).

```
1760001 #include <kernel/net.h>
1760002 #include <kernel/net/route.h>
1760003 #include <kernel/net/arp.h>
1760004 #include <errno.h>
1760005 #include <kernel/proc.h>
1760006 //-----
1760007 // This syscall is present only inside os32.
1760008 //-----
1760009 int
1760010 s_ipconfig (pid_t pid, int n, in_addr_t addr, int m)
1760011 {
1760012     //
1760013     // Must be a privileged process.
1760014     //
1760015     if (proc_table[pid].euid != 0)
1760016     {
```



```
1760017     errset (EPERM);
1760018     return (-1);
1760019 }
1760020 //
1760021 // Check arguments: net0 cannot be modified, because
1760022 // it is necessarily
1760023 // assigned to the loopback virtual interface.
1760024 //
1760025 if (n > NET_MAX_DEVICES || n < 1)
1760026 {
1760027     errset (EINVAL);
1760028     return (-1);
1760029 }
1760030 if (m > 32 || n < 0)
1760031 {
1760032     errset (EINVAL);
1760033     return (-1);
1760034 }
1760035 //
1760036 // Verify that the interface is present.
1760037 //
1760038 if (net_table[n].type == NET_DEV_NULL)
1760039 {
1760040     errset (ENODEV);
1760041     return (-1);
1760042 }
1760043 //
1760044 // Remove previous route related to the interface,
1760045 // if the interface
1760046 // was already configured.
1760047 //
1760048 if (net_table[n].ip != 0 && net_table[n].m != 0)
1760049 {
1760050     s_routedel (pid, net_table[n].ip, net_table[n].m);
1760051 }
1760052 //
1760053 // Modify the net_table[].
```

```
1760054 //
1760055 net_table[n].ip = ntohl (addr);
1760056 net_table[n].m = m;
1760057 //
1760058 // Add the route.
1760059 //
1760060 if (net_table[n].ip != 0 && net_table[n].m != 0)
1760061 {
1760062     return (s_routeadd (pid, addr, m, 0, n));
1760063 }
1760064 //
1760065 return (0);
1760066 }
```

94.8.19 kernel/lib_s/s_kill.c



Si veda la sezione [87.29](#).

```
1770001 #include <kernel/proc.h>
1770002 #include <kernel/lib_s.h>
1770003 #include <errno.h>
1770004 //-----
1770005 int
1770006 s_kill (pid_t pid_killer, pid_t pid_target, int sig)
1770007 {
1770008     uid_t euid = proc_table[pid_killer].euid;
1770009     uid_t uid = proc_table[pid_killer].uid;
1770010     pid_t pgrp = proc_table[pid_killer].pgrp;
1770011     int p;          // Index inside the process table.
1770012     //
1770013     if (pid_target < -1)
1770014     {
1770015         errset (ESRCH);
1770016         return (-1);
1770017     }
1770018     else if (pid_target == -1)
1770019     {
```

```
1770020     if (sig == 0)
1770021     {
1770022         return (0);
1770023     }
1770024     if (euid == 0)
1770025     {
1770026         //
1770027         // Because 'pid_target' is qual to '-1' and
1770028         // the effective
1770029         // user identity is '0', then, all
1770030         // processes,
1770031         // except the kernel and init, will receive
1770032         // the signal.
1770033         //
1770034         // The following scan starts from 2, to
1770035         // preserve the
1770036         // kernel and init processes.
1770037         //
1770038         for (p = 2; p < PROCESS_MAX; p++)
1770039         {
1770040             if (proc_table[p].status != PROC_EMPTY
1770041                 && proc_table[p].status != PROC_ZOMBIE)
1770042             {
1770043                 proc_sig_on (p, sig);
1770044             }
1770045         }
1770046     }
1770047     else
1770048     {
1770049         //
1770050         // Because 'pid_target' is qual to '-1', but
1770051         // the effective
1770052         // user identity is not '0', then, all
1770053         // processes owned
1770054         // by the same effective user identity, will
1770055         // receive the
1770056         // signal.
```

```
1770057 //
1770058 // The following scan starts from 1, to
1770059 // preserve the
1770060 // kernel process.
1770061 //
1770062 for (p = 1; p < PROCESS_MAX; p++)
1770063 {
1770064     if (proc_table[p].status != PROC_EMPTY
1770065         && proc_table[p].status !=
1770066         PROC_ZOMBIE && proc_table[p].uid == euid)
1770067     {
1770068         proc_sig_on (p, sig);
1770069     }
1770070 }
1770071 }
1770072 return (0);
1770073 }
1770074 else if (pid_target == 0)
1770075 {
1770076     if (sig == 0)
1770077     {
1770078         return (0);
1770079     }
1770080 //
1770081 // The following scan starts from 1, to preserve
1770082 // the
1770083 // kernel process.
1770084 //
1770085 for (p = 1; p < PROCESS_MAX; p++)
1770086 {
1770087     if (proc_table[p].status != PROC_EMPTY
1770088         && proc_table[p].status != PROC_ZOMBIE
1770089         && proc_table[p].pgrp == pgrp)
1770090     {
1770091         proc_sig_on (p, sig);
1770092     }
1770093 }
```

```
1770094     return (0);
1770095 }
1770096 else if (pid_target >= PROCESS_MAX)
1770097 {
1770098     errset (ESRCH);
1770099     return (-1);
1770100 }
1770101 else // (pid_target > 0)
1770102 {
1770103     if (proc_table[pid_target].status == PROC_EMPTY
1770104         || proc_table[pid_target].status == PROC_ZOMBIE)
1770105     {
1770106         errset (ESRCH);
1770107         return (-1);
1770108     }
1770109     else if (uid == proc_table[pid_target].uid ||
1770110             uid == proc_table[pid_target].suid ||
1770111             euid == proc_table[pid_target].uid ||
1770112             euid == proc_table[pid_target].suid
1770113             || euid == 0)
1770114     {
1770115         if (sig == 0)
1770116         {
1770117             return (0);
1770118         }
1770119         else
1770120         {
1770121             proc_sig_on (pid_target, sig);
1770122             return (0);
1770123         }
1770124     }
1770125     else
1770126     {
1770127         errset (EPERM);
1770128         return (-1);
1770129     }
1770130 }
```

1770131

}

94.8.20 kernel/lib_s/s_link.c

<<

Si veda la sezione [87.30](#).

```
1780001 #include <kernel/fs.h>
1780002 #include <errno.h>
1780003 #include <kernel/proc.h>
1780004 #include <kernel/lib_s.h>
1780005 //-----
1780006 int
1780007 s_link (pid_t pid, const char *path_old,
1780008         const char *path_new)
1780009 {
1780010     proc_t *ps;
1780011     inode_t *inode_old;
1780012     inode_t *inode_new;
1780013     char path_new_full[PATH_MAX];
1780014     //
1780015     // Get process.
1780016     //
1780017     ps = proc_reference (pid);
1780018     //
1780019     // Try to get the old path inode.
1780020     //
1780021     inode_old = path_inode (pid, path_old);
1780022     if (inode_old == NULL)
1780023     {
1780024         //
1780025         // Cannot get the inode: 'errno' is already set
1780026         // by
1780027         // 'path_inode()'.
1780028         //
1780029         errset (errno);
1780030         return (-1);
1780031     }
```

```
1780032 //
1780033 // The inode is available and checks are done:
1780034 // arrange to get a
1780035 // packed full path name and then the destination
1780036 // directory path.
1780037 //
1780038 path_full (path_new, ps->path_cwd, path_new_full);
1780039 //
1780040 //
1780041 //
1780042 inode_new =
1780043     path_inode_link (pid, path_new_full, inode_old,
1780044                     (mode_t) 0);
1780045 if (inode_new == NULL)
1780046     {
1780047     inode_put (inode_old);
1780048     return (-1);
1780049     }
1780050 if (inode_new != inode_old)
1780051     {
1780052     inode_put (inode_new);
1780053     inode_put (inode_old);
1780054     errset (EUNKNOWN);           // Unknown error.
1780055     return (-1);
1780056     }
1780057 //
1780058 // Inode data is already updated by
1780059 // 'path_inode_link()': just put
1780060 // it and return. Please note that only one is put,
1780061 // because it is
1780062 // just the same of the other.
1780063 //
1780064 inode_put (inode_new);
1780065 return (0);
1780066 }
```

94.8.21 kernel/lib_s/s_listen.c



Si veda la sezione [87.31](#).

```
1790001 #include <kernel/proc.h>
1790002 #include <kernel/lib_s.h>
1790003 #include <kernel/lib_k.h>
1790004 #include <errno.h>
1790005 #include <fcntl.h>
1790006 #include <sys/socket.h>
1790007 #include <arpa/inet.h>
1790008 //-----
1790009 int
1790010 s_listen (pid_t pid, int sfdn, int backlog)
1790011 {
1790012     fd_t *sfd;
1790013     int s;
1790014     //
1790015     // Get file descriptor and verify that it is a
1790016     // socket.
1790017     //
1790018     sfd = fd_reference (pid, &sfdn);
1790019     if (sfd == NULL || sfd->file == NULL)
1790020     {
1790021         errset (EBADF);    // Bad file descriptor.
1790022         return (-1);
1790023     }
1790024     if (sfd->file->sock == NULL)
1790025     {
1790026         errset (ENOTSOCK);    // Not a socket.
1790027         return (-1);
1790028     }
1790029     if (sfd->file->sock->type != SOCK_STREAM)
1790030     {
1790031         errset (EOPNOTSUPP);    // Not a stream
1790032         // socket.
1790033         return (-1);
1790034     }
```



```
1790035     if (sfd->file->sock->raddr != 0
1790036         || sfd->file->sock->rport != 0)
1790037     {
1790038         //
1790039         // The socket is connected, and cannot be good
1790040         // for
1790041         // listening.
1790042         //
1790043         errset (EISCONN);
1790044         return (-1);
1790045     }
1790046     //
1790047     // Scan the other sockets to find if there is
1790048     // another one listening.
1790049     //
1790050     for (s = 0; s < SOCK_MAX_SLOTS; s++)
1790051     {
1790052         if (sock_table[s].tcp.conn == TCP_LISTEN
1790053             && sock_table[s].lport == sfd->file->sock->lport)
1790054         {
1790055             //
1790056             // Yes, there is one: sorry.
1790057             //
1790058             errset (EADDRINUSE);
1790059             return (-1);
1790060         }
1790061     }
1790062     //
1790063     // Check the current TCP state.
1790064     //
1790065     if (sfd->file->sock->tcp.conn != 0
1790066         && sfd->file->sock->tcp.conn != TCP_LISTEN)
1790067     {
1790068         //
1790069         // Cannot change the socket stream state.
1790070         //
1790071         errset (EISCONN);
```

```
1790072     return (-1);
1790073     }
1790074     //
1790075     // The socket might be already listening, but the
1790076     // newly requested
1790077     // queue should be greater or equal to the previous
1790078     // one.
1790079     //
1790080     if (sfd->file->sock->tcp.conn == TCP_LISTEN
1790081         && backlog < sfd->file->sock->tcp.listen_max)
1790082     {
1790083         //
1790084         // Cannot reduce the listen queue: just ignore.
1790085         //
1790086         return (0);
1790087     }
1790088     //
1790089     // Ok.
1790090     //
1790091     sfd->file->sock->tcp.conn = TCP_LISTEN;
1790092     sfd->file->sock->tcp.listen_max =
1790093         min (backlog, SOCK_MAX_QUEUE);
1790094     sfd->file->sock->tcp.listen_pid = pid;
1790095     return (0);
1790096 }
```

94.8.22 kernel/lib_s/s_longjmp.c

«

Si veda la sezione [87.49](#).

```
1800001 #include <kernel/lib_s.h>
1800002 #include <kernel/proc.h>
1800003 #include <errno.h>
1800004 #include <sys/os32.h>
1800005 //-----
1800006 extern uint32_t proc_stack_pointer;
1800007 //-----
```

```
1800008 void
1800009 s_longjmp (pid_t pid, jmp_buf env, int val)
1800010 {
1800011     jmp_stack_t *sp;
1800012     jmp_env_t *jmpenv;
1800013     //
1800014     // Translate the pointer 'env', to the kernel point
1800015     // of view.
1800016     //
1800017     jmpenv = ptr (pid, env);
1800018     //
1800019     // Find where *was* the process stack in memory,
1800020     // from the kernel point
1800021     // of view. Please notice that the current stack at
1800022     // 'proc_stack_pointer' will be saved from the
1800023     // scheduler inside
1800024     // the process table. So, the replacement is made at
1800025     // the current
1800026     // stack position, and not inside the process table.
1800027     //
1800028     sp = ptr (pid, (void *) jmpenv->esp0);
1800029     //
1800030     // Restore the process stack.
1800031     //
1800032     sp->eax0 = jmpenv->eax0;
1800033     sp->ecx0 = jmpenv->ecx0;
1800034     sp->edx0 = jmpenv->edx0;
1800035     sp->ebx0 = jmpenv->ebx0;
1800036     sp->ebp0 = jmpenv->ebp0;
1800037     sp->esi0 = jmpenv->esi0;
1800038     sp->edi0 = jmpenv->edi0;
1800039     sp->ds0 = jmpenv->ds0;
1800040     sp->es0 = jmpenv->es0;
1800041     sp->fs0 = jmpenv->fs0;
1800042     sp->gs0 = jmpenv->gs0;
1800043     sp->eflags0 = jmpenv->eflags0;
1800044     sp->cs0 = jmpenv->cs0;
```

```
1800045     sp->eip0 = jmpenv->eip0;
1800046     //
1800047     sp->eip1 = jmpenv->eip1;
1800048     sp->syscallnr = jmpenv->syscallnr;
1800049     sp->msg_pointer = jmpenv->msg_pointer;
1800050     sp->msg_size = jmpenv->msg_size;
1800051     sp->env = jmpenv->env;
1800052     sp->ret = val;
1800053     sp->ebp1 = jmpenv->ebp1;
1800054     sp->eip2 = jmpenv->eip2;
1800055     //
1800056     // Replace the stack pointer too!
1800057     //
1800058     proc_stack_pointer = jmpenv->esp0;
1800059 }
```

94.8.23 kernel/lib_s/s_lseek.c



Si veda la sezione [87.33](#).

```
1810001 #include <kernel/proc.h>
1810002 #include <kernel/lib_s.h>
1810003 #include <errno.h>
1810004 //-----
1810005 off_t
1810006 s_lseek (pid_t pid, int fdn, off_t offset, int whence)
1810007 {
1810008     inode_t *inode;
1810009     file_t *file;
1810010     fd_t *fd;
1810011     off_t test_offset;
1810012     //
1810013     // Get file descriptor.
1810014     //
1810015     fd = fd_reference (pid, &fdn);
1810016     if (fd == NULL || fd->file == NULL
1810017         || fd->file->inode == NULL)
```

```
1810018     {
1810019         errset (EBADF);    // Bad file descriptor.
1810020         return (-1);
1810021     }
1810022     //
1810023     // Get file table item.
1810024     //
1810025     file = fd->file;
1810026     //
1810027     // Get inode.
1810028     //
1810029     inode = file->inode;
1810030     //
1810031     // Change position depending on the 'whence'
1810032     // parameter.
1810033     //
1810034     if (whence == SEEK_SET)
1810035     {
1810036         if (offset < 0)
1810037         {
1810038             errset (EINVAL);    // Invalid argument.
1810039             return ((off_t) - 1);
1810040         }
1810041         else
1810042         {
1810043             fd->file->offset = offset;
1810044         }
1810045     }
1810046     else if (whence == SEEK_CUR)
1810047     {
1810048         test_offset = fd->file->offset;
1810049         test_offset += offset;
1810050         if (test_offset < 0)
1810051         {
1810052             errset (EINVAL);    // Invalid argument.
1810053             return ((off_t) - 1);
1810054         }
    }
```

```
1810055     else
1810056     {
1810057         fd->file->offset = test_offset;
1810058     }
1810059 }
1810060 else if (whence == SEEK_END)
1810061 {
1810062     test_offset = inode->size;
1810063     test_offset += offset;
1810064     if (test_offset < 0)
1810065     {
1810066         errset (EINVAL);    // Invalid argument.
1810067         return ((off_t) - 1);
1810068     }
1810069     else
1810070     {
1810071         fd->file->offset = test_offset;
1810072     }
1810073 }
1810074 else
1810075 {
1810076     errset (EINVAL); // Invalid argument.
1810077     return ((off_t) - 1);
1810078 }
1810079 //
1810080 // Return the new file position.
1810081 //
1810082 return (fd->file->offset);
1810083 }
```

94.8.24 kernel/lib_s/s_mkdir.c



Si veda la sezione [87.34](#).

```
1820001 #include <kernel/fs.h>
1820002 #include <errno.h>
1820003 #include <kernel/proc.h>
```

```
1820004 #include <libgen.h>
1820005 #include <kernel/lib_s.h>
1820006 //-----
1820007 int
1820008 s_mkdir (pid_t pid, const char *path, mode_t mode)
1820009 {
1820010     proc_t *ps;
1820011     inode_t *inode_directory;
1820012     inode_t *inode_parent;
1820013     int status;
1820014     char path_directory[PATH_MAX];
1820015     char path_copy[PATH_MAX];
1820016     char *path_parent;
1820017     ssize_t size_written;
1820018     //
1820019     struct
1820020     {
1820021         ino_t ino_1;
1820022         char name_1[NAME_MAX];
1820023         ino_t ino_2;
1820024         char name_2[NAME_MAX];
1820025     } directory;
1820026     //
1820027     // Get process.
1820028     //
1820029     ps = proc_reference (pid);
1820030     //
1820031     // Correct the mode with the umask.
1820032     //
1820033     mode &= ~ps->umask;
1820034     //
1820035     // Inside 'mode', the file type is fixed. No check
1820036     // is made.
1820037     //
1820038     mode &= 00777;
1820039     mode |= S_IFDIR;
1820040     //
```

```
1820041 // The full path and the directory path is needed.
1820042 //
1820043 status = path_full (path, ps->path_cwd, path_directory);
1820044 if (status < 0)
1820045     {
1820046         return (-1);
1820047     }
1820048 strncpy (path_copy, path_directory, PATH_MAX);
1820049 path_copy[PATH_MAX - 1] = 0;
1820050 path_parent = dirname (path_copy);
1820051 //
1820052 // Check if something already exists with the same
1820053 // name. The scan
1820054 // is done with kernel privileges.
1820055 //
1820056 inode_directory = path_inode ((uid_t) 0, path_directory);
1820057 if (inode_directory != NULL)
1820058     {
1820059         //
1820060         // The file already exists. Put inode and return
1820061         // an error.
1820062         //
1820063         inode_put (inode_directory);
1820064         errset (EEXIST); // File exists.
1820065         return (-1);
1820066     }
1820067 //
1820068 // Try to locate the directory that should contain
1820069 // this one.
1820070 //
1820071 inode_parent = path_inode (pid, path_parent);
1820072 if (inode_parent == NULL)
1820073     {
1820074         //
1820075         // Cannot locate the directory: return an error.
1820076         // The variable
1820077         // 'errno' should already be set by
```



```
1820078         // 'path_inode()'.
1820079         //
1820080         errset (errno);
1820081         return (-1);
1820082     }
1820083     //
1820084     // Try to create the node: should fail if the user
1820085     // does not have
1820086     // enough permissions.
1820087     //
1820088     inode_directory =
1820089         path_inode_link (pid, path_directory, NULL, mode);
1820090     if (inode_directory == NULL)
1820091     {
1820092         //
1820093         // Sorry: cannot create the inode! The variable
1820094         // 'errno' should
1820095         // already be set by 'path_inode_link()'.
1820096         //
1820097         errset (errno);
1820098         return (-1);
1820099     }
1820100     //
1820101     // Fill records for '.' and '..'.
1820102     //
1820103     directory.ino_1 = inode_directory->ino;
1820104     strncpy (directory.name_1, ".", (size_t) 3);
1820105     directory.ino_2 = inode_parent->ino;
1820106     strncpy (directory.name_2, "..", (size_t) 3);
1820107     //
1820108     // Write data.
1820109     //
1820110     size_written =
1820111         inode_file_write (inode_directory, (off_t) 0,
1820112                          &directory, (sizeof directory));
1820113     if (size_written != (sizeof directory))
1820114     {
```

```
1820115     return (-1);
1820116     }
1820117     //
1820118     // Fix directory inode links.
1820119     //
1820120     inode_directory->links = 2;
1820121     inode_directory->time = s_time (pid, NULL);
1820122     inode_directory->changed = 1;
1820123     //
1820124     // Fix parent directory inode links.
1820125     //
1820126     inode_parent->links++;
1820127     inode_parent->time = s_time (pid, NULL);
1820128     inode_parent->changed = 1;
1820129     //
1820130     // Save and put the inodes.
1820131     //
1820132     inode_save (inode_parent);
1820133     inode_save (inode_directory);
1820134     inode_put (inode_parent);
1820135     inode_put (inode_directory);
1820136     //
1820137     // Return.
1820138     //
1820139     return (0);
1820140 }
```

94.8.25 kernel/lib_s/s_mknod.c



Si veda la sezione [87.35](#).

```
1830001 #include <kernel/fs.h>
1830002 #include <errno.h>
1830003 #include <kernel/proc.h>
1830004 #include <kernel/lib_s.h>
1830005 //-----
1830006 int
```

```
1830007 s_mknod (pid_t pid, const char *path, mode_t mode,
1830008         dev_t device)
1830009 {
1830010     proc_t *ps;
1830011     inode_t *inode;
1830012     char full_path[PATH_MAX];
1830013     //
1830014     // Get process.
1830015     //
1830016     ps = proc_reference (pid);
1830017     //
1830018     // Correct the mode with the umask.
1830019     //
1830020     mode &= ~ps->umask;
1830021     //
1830022     // Currently must be root, unless the type is a
1830023     // regular file,
1830024     // or a FIFO file.
1830025     //
1830026     if (!(S_ISFIFO (mode) || S_ISREG (mode)))
1830027     {
1830028         if (ps->uid != 0)
1830029         {
1830030             errset (EPERM);           // Operation not
1830031             // permitted.
1830032             return (-1);
1830033         }
1830034     }
1830035     //
1830036     // Check the type of node requested.
1830037     //
1830038     if (!(S_ISBLK (mode) ||
1830039         S_ISCHR (mode) ||
1830040         S_ISREG (mode) || S_ISFIFO (mode)
1830041         || S_ISDIR (mode)))
1830042     {
1830043         errset (EINVAL); // Invalid argument.
```

```
1830044     return (-1);
1830045     }
1830046     //
1830047     // Check if something already exists with the same
1830048     // name.
1830049     //
1830050     inode = path_inode (pid, path);
1830051     if (inode != NULL)
1830052     {
1830053         //
1830054         // The file already exists. Put inode and return
1830055         // an error.
1830056         //
1830057         inode_put (inode);
1830058         errset (EEXIST); // File exists.
1830059         return (-1);
1830060     }
1830061     //
1830062     // Try to creat the node.
1830063     //
1830064     path_full (path, ps->path_cwd, full_path);
1830065     inode = path_inode_link (pid, full_path, NULL, mode);
1830066     if (inode == NULL)
1830067     {
1830068         //
1830069         // Sorry: cannot create the inode!
1830070         //
1830071         return (-1);
1830072     }
1830073     //
1830074     // Set the device number if necessary.
1830075     //
1830076     if (S_ISBLK (mode) || S_ISCHR (mode))
1830077     {
1830078         inode->direct[0] = device;
1830079         inode->changed = 1;
1830080     }
```

```
1830081 //
1830082 // Put the inode.
1830083 //
1830084 inode_put (inode);
1830085 //
1830086 // Return.
1830087 //
1830088 return (0);
1830089 }
```

94.8.26 kernel/lib_s/s_mount.c



Si veda la sezione [87.36](#).

```
1840001 #include <kernel/fs.h>
1840002 #include <errno.h>
1840003 #include <kernel/proc.h>
1840004 #include <kernel/lib_s.h>
1840005 //-----
1840006 int
1840007 s_mount (pid_t pid, const char *path_dev,
1840008         const char *path_mnt, int options)
1840009 {
1840010     proc_t *ps;
1840011     dev_t device; // Device to mount.
1840012     inode_t *inode_mnt; // Directory mount point.
1840013     void *pstatus;
1840014     //
1840015     // Get process.
1840016     //
1840017     ps = proc_reference (pid);
1840018     //
1840019     // Verify to be the super user.
1840020     //
1840021     if (ps->euid != 0)
1840022     {
1840023         errset (EPERM); // Operation not permitted.
```

```
1840024     return (-1);
1840025     }
1840026     //
1840027     device = path_device (pid, path_dev);
1840028     if (device < 0)
1840029     {
1840030         return (-1);
1840031     }
1840032     //
1840033     inode_mnt = path_inode (pid, path_mnt);
1840034     if (inode_mnt == NULL)
1840035     {
1840036         return (-1);
1840037     }
1840038     if (!S_ISDIR (inode_mnt->mode))
1840039     {
1840040         inode_put (inode_mnt);
1840041         errset (ENOTDIR); // Not a directory.
1840042         return (-1);
1840043     }
1840044     if (inode_mnt->sb_attached != NULL)
1840045     {
1840046         inode_put (inode_mnt);
1840047         errset (EBUSY); // Device or resource busy.
1840048         return (-1);
1840049     }
1840050     //
1840051     // All data is available.
1840052     //
1840053     pstatus = sb_mount (device, &inode_mnt, options);
1840054     if (pstatus == NULL)
1840055     {
1840056         inode_put (inode_mnt);
1840057         return (-1);
1840058     }
1840059     //
1840060     return (0);
```

1840061

}

94.8.27 kernel/lib_s/s_open.c

Si veda la sezione [87.37](#).

```
1850001 #include <kernel/proc.h>
1850002 #include <kernel/lib_s.h>
1850003 #include <kernel/lib_k.h>
1850004 #include <errno.h>
1850005 #include <fcntl.h>
1850006 //-----
1850007 int
1850008 s_open (pid_t pid, const char *path, int oflags,
1850009         mode_t mode)
1850010 {
1850011     inode_t *inode;
1850012     int status;
1850013     file_t *file;
1850014     fd_t *fd;
1850015     int fdn;
1850016     char full_path[PATH_MAX];
1850017     int perm;
1850018     tty_t *tty;
1850019     mode_t umask;
1850020     int errno_save;
1850021     //
1850022     // k_printf ("%s(%i, %s, %x, %05o)\n", __func__,
1850023     // (int) pid,
1850024     // path, oflags, (int) mode);
1850025     //
1850026     // Check path argument.
1850027     //
1850028     if (path == NULL || strlen (path) == 0)
1850029     {
1850030         errset (EINVAL); // Invalid argument.
1850031         return (-1);
```

```
1850032     }
1850033     //
1850034     // Correct the mode with the umask. As it is not a
1850035     // directory, to the
1850036     // mode are removed execution and sticky
1850037     // permissions.
1850038     //
1850039     umask = proc_table[pid].umask | 01111;
1850040     mode &= ~umask;
1850041     //
1850042     // Check open options.
1850043     //
1850044     if (oflags & O_WRONLY)
1850045     {
1850046         //
1850047         // The file is to be opened for write, or for
1850048         // read/write.
1850049         // Try to get inode.
1850050         //
1850051         inode = path_inode (pid, path);
1850052         if (inode == NULL)
1850053         {
1850054             //
1850055             // Cannot get the inode. See if there is the
1850056             // creation
1850057             // option.
1850058             //
1850059             if (oflags & O_CREAT)
1850060             {
1850061                 //
1850062                 // Try to create the missing inode: the
1850063                 // file must be a
1850064                 // regular one, so add the mode.
1850065                 //
1850066                 path_full (path,
1850067                             proc_table[pid].path_cwd,
1850068                             full_path);
```



```
1850069         inode =
1850070             path_inode_link (pid, full_path, NULL,
1850071                             (mode | S_IFREG));
1850072         if (inode == NULL)
1850073             {
1850074                 //
1850075                 // Sorry: cannot create the inode!
1850076                 // Variable 'errno'
1850077                 // is already set by
1850078                 // 'path_inode_link()'.
1850079                 //
1850080                 errset (errno);
1850081                 return (-1);
1850082             }
1850083     }
1850084     else
1850085     {
1850086         //
1850087         // Cannot open the inode. Variable
1850088         // 'errno'
1850089         // should be already set by
1850090         // 'path_inode()'.
1850091         //
1850092         errset (errno);
1850093         return (-1);
1850094     }
1850095 }
1850096 //
1850097 // The inode was read or created: check if it
1850098 // must be
1850099 // truncated. It can be truncated only if it is
1850100 // a regular
1850101 // file.
1850102 //
1850103 if (oflags & O_TRUNC && inode->mode & S_IFREG)
1850104     {
1850105         //
```

```
1850106         // Truncate inode.
1850107         //
1850108         status = inode_truncate (inode);
1850109         if (status != 0)
1850110             {
1850111                 //
1850112                 // Cannot truncate the inode: release it
1850113                 // and return.
1850114                 // But this error should never happen,
1850115                 // because the
1850116                 // function 'inode_truncate()' will not
1850117                 // return any
1850118                 // other value than zero.
1850119                 //
1850120                 errno_save = errno;
1850121                 inode_put (inode);
1850122                 errset (errno_save);
1850123                 return (-1);
1850124             }
1850125     }
1850126 }
1850127 else
1850128     {
1850129         //
1850130         // The file is to be opened for read, but not
1850131         // for write.
1850132         // Try to get inode.
1850133         //
1850134         inode = path_inode (pid, path);
1850135         if (inode == NULL)
1850136             {
1850137                 //
1850138                 // Cannot open the file.
1850139                 //
1850140                 errset (errno);
1850141                 return (-1);
1850142             }
```

```
1850143     }
1850144     //
1850145     // An inode was opened: check type and access
1850146     // permissions.
1850147     // All file types are good, even directories, as the
1850148     // type
1850149     // DIR is implemented through file descriptors.
1850150     //
1850151     perm = 0;
1850152     if (oflags & O_RDONLY)
1850153         perm |= 4;
1850154     if (oflags & O_WRONLY)
1850155         perm |= 2;
1850156     status =
1850157         inode_check (inode, S_IFMT, perm,
1850158                     proc_table[pid].euid,
1850159                     proc_table[pid].egid);
1850160     if (status != 0)
1850161     {
1850162         //
1850163         // The file type is not correct or the user does
1850164         // not have
1850165         // permissions.
1850166         //
1850167         return (-1);
1850168     }
1850169     //
1850170     // Allocate the file, inside the file table.
1850171     //
1850172     file = file_reference (-1);
1850173     if (file == NULL)
1850174     {
1850175         //
1850176         // Cannot allocate the file inside the file
1850177         // table: release the
1850178         // inode, update 'errno' and return.
1850179         //
```

```
1850180     inode_put (inode);
1850181     errset (ENFILE); // Too many files open in
1850182     // system.
1850183     return (-1);
1850184 }
1850185 //
1850186 // Put some data inside the file item. Only options
1850187 // O_RDONLY and O_WRONLY are kept here, because the
1850188 // O_APPEND
1850189 // is saved inside the file descriptor table.
1850190 //
1850191 file->references = 1;
1850192 file->oflags = (oflags & (O_RDONLY | O_WRONLY));
1850193 file->inode = inode;
1850194 file->sock = NULL;
1850195 //
1850196 // Allocate the file descriptor: variable 'fdn' will
1850197 // be modified
1850198 // by the call to 'fd_reference()'.
1850199 //
1850200 fdn = -1;
1850201 fd = fd_reference (pid, &fdn);
1850202 if (fd == NULL)
1850203     {
1850204         //
1850205         // Cannot allocate the file descriptor: remove
1850206         // the item from
1850207         // file table.
1850208         //
1850209         file->references = 0;
1850210         file->oflags = 0;
1850211         file->inode = NULL;
1850212         file->sock = NULL;
1850213         //
1850214         // Release the inode.
1850215         //
1850216         inode_put (inode);
```

```
1850217      //
1850218      // Return an error.
1850219      //
1850220      errset (EMFILE); // Too many open files.
1850221      return (-1);
1850222  }
1850223  //
1850224  // File descriptor allocated: put some data inside
1850225  // the
1850226  // file descriptor item.
1850227  //
1850228  fd->fl_flags =
1850229      (oflags & (O_RDONLY | O_WRONLY | O_APPEND));
1850230  fd->fd_flags = 0;
1850231  fd->file = file;
1850232  fd->file->offset = 0;
1850233  //
1850234  // Check for particular types and situations.
1850235  //
1850236  if ((S_ISCHR (inode->mode))
1850237      && (oflags & O_RDONLY) && (oflags & O_WRONLY))
1850238  {
1850239      //
1850240      // The inode is a character special file
1850241      // (related to a character
1850242      // device), opened for read and write!
1850243      //
1850244      if ((inode->direct[0] & 0xFF00) ==
1850245          (DEV_CONSOLE_MAJOR << 8))
1850246      {
1850247          //
1850248          // It is a terminal (currently only consoles
1850249          // are possible).
1850250          // Get the tty reference.
1850251          //
1850252          tty = tty_reference ((dev_t) inode->direct[0]);
1850253          //
```

```
1850254 // Verify that the terminal is not already
1850255 // the controlling
1850256 // terminal of some process group.
1850257 //
1850258 if (tty->pgrp == 0)
1850259 {
1850260 //
1850261 // The terminal is free: verify if the
1850262 // current process
1850263 // needs a controlling terminal.
1850264 //
1850265 if (proc_table[pid].device_tty == 0
1850266     && proc_table[pid].pgrp == pid)
1850267 {
1850268 //
1850269 // It is a group leader with no
1850270 // controlling
1850271 // terminal: set the controlling
1850272 // terminal.
1850273 //
1850274 proc_table[pid].device_tty =
1850275     inode->direct[0];
1850276 tty->pgrp = proc_table[pid].pgrp;
1850277 }
1850278 }
1850279 }
1850280 }
1850281 else if (S_ISFIFO (inode->mode))
1850282 {
1850283 //
1850284 // It is FIFO (named pipe).
1850285 //
1850286 if ((oflags & O_ACCMODE) == O_RDWR)
1850287 {
1850288     inode->pipe_ref_read++;
1850289     inode->pipe_ref_write++;
1850290 }
```

```
1850291     else if (oflags & O_RDONLY)
1850292     {
1850293         inode->pipe_ref_read++;
1850294         //
1850295         // Go to sleep if there are no processes
1850296         // writing to the
1850297         // inode. Otherwise, wake them up.
1850298         //
1850299         if (inode->pipe_ref_write == 0)
1850300         {
1850301             proc_table[pid].status = PROC_SLEEPING;
1850302             proc_table[pid].ret = 0;
1850303             proc_table[pid].wakeup_inode = inode;
1850304             proc_table[pid].wakeup_events =
1850305                 WAKEUP_EVENT_PIPE_READ;
1850306         }
1850307         else
1850308         {
1850309             proc_wakeup_pipe_write (inode);
1850310         }
1850311     }
1850312     else if (oflags & O_WRONLY)
1850313     {
1850314         inode->pipe_ref_write++;
1850315         //
1850316         // Go to sleep if there are no processes
1850317         // reading to the
1850318         // inode. Otherwise, wake them up.
1850319         //
1850320         if (inode->pipe_ref_read == 0)
1850321         {
1850322             proc_table[pid].status = PROC_SLEEPING;
1850323             proc_table[pid].ret = 0;
1850324             proc_table[pid].wakeup_inode = inode;
1850325             proc_table[pid].wakeup_events =
1850326                 WAKEUP_EVENT_PIPE_WRITE;
1850327         }
```

```
1850328         else
1850329             {
1850330                 proc_wakeup_pipe_read (inode);
1850331             }
1850332     }
1850333 }
1850334 //
1850335 // Return the file descriptor.
1850336 //
1850337 return (fdn);
1850338 }
```

94.8.28 kernel/lib_s/s_pipe.c

«

Si veda la sezione [87.38](#).

```
1860001 #include <kernel/proc.h>
1860002 #include <kernel/lib_s.h>
1860003 #include <kernel/lib_k.h>
1860004 #include <errno.h>
1860005 #include <fcntl.h>
1860006 //-----
1860007 int
1860008 s_pipe (pid_t pid, int pipefd[2])
1860009 {
1860010     file_t *file;
1860011     fd_t *fd_read;
1860012     fd_t *fd_write;
1860013     int fdn_read;
1860014     int fdn_write;
1860015     //
1860016     // Allocate the file inside the file table and the
1860017     // inode inside
1860018     // the inode table.
1860019     //
1860020     file = file_pipe_make ();
1860021     if (file == NULL)
```



```
1860022     {
1860023         errset (errno);
1860024         return (-1);
1860025     }
1860026     //
1860027     // Prepare file descriptor for read.
1860028     //
1860029     fdn_read = -1;
1860030     fd_read = fd_reference (pid, &fdn_read);
1860031     if (fd_read == NULL)
1860032     {
1860033         //
1860034         // Cannot allocate the file descriptor: remove
1860035         // the item from
1860036         // file table and put the relative inode.
1860037         //
1860038         file->references = 0;
1860039         file->oflags = 0;
1860040         inode_put (file->inode);
1860041         file->inode = NULL;
1860042         //
1860043         // Return an error.
1860044         //
1860045         errset (EMFILE); // Too many open files.
1860046         return (-1);
1860047     }
1860048     //
1860049     // File descriptor allocated: put some data inside
1860050     // the
1860051     // file descriptor item and increment the pipe
1860052     // references
1860053     // for read.
1860054     //
1860055     fd_read->fl_flags = O_RDONLY;
1860056     fd_read->fd_flags = 0;
1860057     fd_read->file = file;
1860058     fd_read->file->offset = 0;
```

```
1860059 fd_read->file->inode->pipe_ref_read++;
1860060 //
1860061 // Prepare file descriptor for write.
1860062 //
1860063 fdn_write = -1;
1860064 fd_write = fd_reference (pid, &fdn_write);
1860065 if (fd_write == NULL)
1860066     {
1860067         //
1860068         // Cannot allocate the file descriptor: remove
1860069         // the item from
1860070         // file table and put the relative inode.
1860071         //
1860072         file->references = 0;
1860073         file->oflags = 0;
1860074         inode_put (file->inode);
1860075         file->inode = NULL;
1860076         //
1860077         // Remove file descriptor for read.
1860078         //
1860079         fd_read->file->inode->pipe_ref_read--;
1860080         fd_read->fl_flags = 0;
1860081         fd_read->fd_flags = 0;
1860082         fd_read->file = NULL;
1860083         //
1860084         // Return an error.
1860085         //
1860086         errset (EMFILE); // Too many open files.
1860087         return (-1);
1860088     }
1860089 //
1860090 // File descriptor allocated: put some data inside
1860091 // the
1860092 // file descriptor item.
1860093 //
1860094 fd_write->fl_flags = O_WRONLY;
1860095 fd_write->fd_flags = 0;
```

```
1860096     fd_write->file = file;
1860097     fd_write->file->offset = 0;
1860098     fd_write->file->inode->pipe_ref_write++;
1860099     //
1860100     // Save file descriptor numbers inside the
1860101     // 'pipefd[]' array.
1860102     //
1860103     pipefd[0] = fdn_read;
1860104     pipefd[1] = fdn_write;
1860105     //
1860106     // Ok.
1860107     //
1860108     return (0);
1860109 }
```

94.8.29 kernel/lib_s/s_read.c

Si veda la sezione [87.39](#).

```
1870001 #include <kernel/proc.h>
1870002 #include <kernel/lib_s.h>
1870003 #include <errno.h>
1870004 #include <fcntl.h>
1870005 //-----
1870006 #define DEBUG 0
1870007 //-----
1870008 ssize_t
1870009 s_read (pid_t pid, int fdn, void *buffer, size_t count)
1870010 {
1870011     fd_t *fd;
1870012     ssize_t size_read;
1870013     int eof = 0;
1870014     //
1870015     // Get file descriptor.
1870016     //
1870017     fd = fd_reference (pid, &fdn);
1870018     if (fd == NULL || fd->file == NULL
```

```
1870019     || (fd->file->inode == NULL
1870020         && fd->file->sock == NULL))
1870021     {
1870022         errset (EBADF);    // Bad file descriptor.
1870023         return ((ssize_t) - 1);
1870024     }
1870025     //
1870026     // Check if it is opened for read.
1870027     //
1870028     if (!(fd->file->oflags & O_RDONLY))
1870029     {
1870030         //
1870031         // The file is not opened for read.
1870032         //
1870033         errset (EINVAL);  // Invalid argument.
1870034         return ((ssize_t) - 1);
1870035     }
1870036     //
1870037     // Check the kind of file to be read and read it.
1870038     //
1870039     if (fd->file->sock != NULL)
1870040     {
1870041         //
1870042         // Read from the socket and return.
1870043         //
1870044         return (s_recvfrom
1870045             (pid, fdn, buffer, count, 0, NULL, NULL));
1870046     }
1870047     else if (S_ISBLK (fd->file->inode->mode)
1870048             || S_ISCHR (fd->file->inode->mode))
1870049     {
1870050         //
1870051         // A device is to be read.
1870052         //
1870053         size_read =
1870054             dev_io (pid,
1870055                 (dev_t) fd->file->inode->direct[0],
```

```
1870056         DEV_READ, fd->file->offset, buffer,
1870057         count, &eof);
1870058     if (size_read < 0
1870059         && (errno == EAGAIN || errno == EWOULDBLOCK))
1870060     {
1870061         if (fd->fl_flags & O_NONBLOCK)
1870062         {
1870063             //
1870064             // Non blocking null read.
1870065             //
1870066             ;
1870067         }
1870068     else
1870069     {
1870070         //
1870071         // Null read: put the process to sleep.
1870072         //
1870073         proc_table[pid].status = PROC_SLEEPING;
1870074         proc_table[pid].ret = 0;
1870075         proc_table[pid].wakeup_events =
1870076             WAKEUP_EVENT_DEV_READ;
1870077         proc_table[pid].wakeup_dev =
1870078             fd->file->inode->direct[0];
1870079         if (DEBUG)
1870080         {
1870081             k_printf
1870082                 ("%s] PID %i goes to sleep "
1870083                 "waiting to read from a "
1870084                 "device.\n", __FILE__, pid);
1870085         }
1870086     }
1870087 }
1870088 }
1870089 else if (S_ISREG (fd->file->inode->mode))
1870090 {
1870091     //
1870092     // A regular file is to be read.
```

```
1870093     //
1870094     size_read =
1870095         inode_file_read (fd->file->inode,
1870096                         fd->file->offset, buffer,
1870097                         count, &eof);
1870098     }
1870099     else if (S_ISDIR (fd->file->inode->mode))
1870100     {
1870101         //
1870102         // A directory, is to be read.
1870103         //
1870104         size_read =
1870105             inode_file_read (fd->file->inode,
1870106                             fd->file->offset, buffer,
1870107                             count, &eof);
1870108     }
1870109     else if (S_ISFIFO (fd->file->inode->mode))
1870110     {
1870111         //
1870112         // A pipe, is to be read.
1870113         //
1870114         size_read =
1870115             inode_pipe_read (fd->file->inode, buffer,
1870116                             count, &eof);
1870117         //
1870118         if (size_read == 0)
1870119             {
1870120                 //
1870121                 // Check what to do.
1870122                 //
1870123                 if (fd->file->inode->pipe_ref_write == 0)
1870124                     {
1870125                         //
1870126                         // EOF, if it is a valid pointer, is
1870127                         // already
1870128                         // set by 'inode_pipe_read()', if is
1870129                         // time to
```

```
1870130         // set it.
1870131         //
1870132         // Wake up processes waiting to write.
1870133         //
1870134         proc_wakeup_pipe_write (fd->file->inode);
1870135         //
1870136         return (size_read);
1870137     }
1870138     else
1870139     {
1870140         //
1870141         // Go to sleep.
1870142         //
1870143         proc_table[pid].status = PROC_SLEEPING;
1870144         proc_table[pid].ret = 0;
1870145         proc_table[pid].wakeup_inode =
1870146             fd->file->inode;
1870147         proc_table[pid].wakeup_events =
1870148             WAKEUP_EVENT_PIPE_READ;
1870149         if (DEBUG)
1870150             {
1870151                 k_printf
1870152                     ("%s] PID %i goes to sleep "
1870153                     "waiting to read from a pipe.\n",
1870154                     __FILE__, pid);
1870155             }
1870156     }
1870157 }
1870158 else
1870159 {
1870160     //
1870161     // Wake up processes waiting to write.
1870162     //
1870163     proc_wakeup_pipe_write (fd->file->inode);
1870164 }
1870165 }
1870166 else
```

```
1870167     {
1870168         //
1870169         // Unsupported file type.
1870170         //
1870171         errset (E_FILE_TYPE_UNSUPPORTED); // File type
1870172         // unsupported.
1870173         return ((ssize_t) - 1);
1870174     }
1870175     //
1870176     // Update the file descriptor internal offset, if
1870177     // there is an inode.
1870178     //
1870179     if (fd->file->inode != NULL && size_read > 0)
1870180     {
1870181         fd->file->offset += size_read;
1870182     }
1870183     //
1870184     // Return the size read, even if it is an error.
1870185     // Please notice
1870186     // that a size of zero might be related to an end of
1870187     // file, or
1870188     // just that the read should be retried. For the
1870189     // latter case,
1870190     // -1 is returned with error EAGAIN, so that the
1870191     // function
1870192     // 'read()' can retry.
1870193     //
1870194     if (size_read == 0 && !eof)
1870195     {
1870196         errset (EAGAIN);
1870197         return (-1);
1870198     }
1870199     else
1870200     {
1870201         return (size_read);
1870202     }
```


1870203

}

94.8.30 kernel/lib_s/s_recvfrom.c

Si veda la sezione [87.40](#).

```
1880001 #include <fcntl.h>
1880002 #include <kernel/proc.h>
1880003 #include <kernel/net.h>
1880004 #include <kernel/net/route.h>
1880005 #include <errno.h>
1880006 #include <arpa/inet.h>
1880007 #include <sys/os32.h>
1880008 #include <netinet/udp.h>
1880009 //-----
1880010 #define DEBUG 0
1880011 //-----
1880012 ssize_t
1880013 s_recvfrom (pid_t pid, int sfdn, void *buffer,
1880014             size_t length, int flags,
1880015             struct sockaddr *addrfrom, socklen_t * addrlen)
1880016 {
1880017     fd_t *sfd;
1880018     int i;           // IP table index.
1880019     size_t size_read;
1880020     struct udphdr *udp;
1880021     void *data;
1880022     struct sockaddr_in *addrfrom_in = (void *) addrfrom;
1880023     //
1880024     // Get file descriptor and verify that it is a
1880025     // socket.
1880026     //
1880027     sfd = fd_reference (pid, &sfdn);
1880028     if (sfd == NULL || sfd->file == NULL)
1880029     {
1880030         errset (EBADF); // Bad file descriptor.
1880031         return ((ssize_t) - 1);

```

```
1880032     }
1880033     if (sfd->file->sock == NULL)
1880034     {
1880035         errset (ENOTSOCK);           // Not a socket.
1880036         return ((ssize_t) - 1);
1880037     }
1880038     //
1880039     // Verify to have a valid buffer pointer.
1880040     //
1880041     if (buffer == NULL)
1880042     {
1880043         errset (EINVAL);
1880044         return ((ssize_t) - 1);
1880045     }
1880046     //
1880047     //
1880048     //
1880049     if (sfd->file->sock->family == AF_INET)
1880050     {
1880051         //
1880052         // INET
1880053         //
1880054         // Should do some check here...
1880055         //
1880056         if (sfd->file->sock->type == SOCK_RAW)
1880057         {
1880058             //
1880059             // RAW
1880060             //
1880061             if (sfd->file->sock->protocol == IPPROTO_ICMP)
1880062             {
1880063                 //
1880064                 // ICMP
1880065                 //
1880066                 //
1880067                 // Scan the ip_table[] to find an ICMP
1880068                 // packet
```

```
1880069 // that was not already seen by the
1880070 // socket.
1880071 //
1880072 for (i = 0; i < IP_MAX_PACKETS; i++)
1880073 {
1880074 //
1880075 // Check the protocol.
1880076 //
1880077 if (ip_table[i].packet.header.protocol !=
1880078     IPPROTO_ICMP)
1880079 {
1880080 //
1880081 // It is not ICMP.
1880082 //
1880083 continue;
1880084 }
1880085 //
1880086 // Is the packet new for the socket?
1880087 //
1880088 // Please notice that the kernel
1880089 // might be interrupted
1880090 // also between clock tics; so,
1880091 // during a single clock
1880092 // time, a new packet might be
1880093 // reached.
1880094 //
1880095 if (ip_table[i].clock
1880096     < sfd->file->sock->read.clock[i])
1880097 {
1880098 //
1880099 // Already seen or packet too
1880100 // old.
1880101 //
1880102 continue;
1880103 }
1880104 //
1880105 // Verify the IP addresses.
```

```
1880106 //
1880107 if (ip_table[i].packet.header.daddr
1880108     != htonl (sfd->file->sock->laddr)
1880109     && sfd->file->sock->laddr != 0)
1880110     {
1880111         //
1880112         // The local address does not
1880113         // match, and it is
1880114         // not zero.
1880115         //
1880116         continue;
1880117     }
1880118 //
1880119 if (ip_table[i].packet.header.saddr
1880120     != htonl (sfd->file->sock->raddr))
1880121     {
1880122         //
1880123         // The remote address does not
1880124         // match, but
1880125         // if it is zero, we accept all.
1880126         //
1880127         if (sfd->file->sock->raddr == 0)
1880128             {
1880129                 //
1880130                 // Can accept the packet.
1880131                 //
1880132                 ;
1880133             }
1880134         else
1880135             {
1880136                 continue;
1880137             }
1880138     }
1880139 //
1880140 // Packet accepted.
1880141 //
1880142 // This ICMP RAW packet is new for
```

```
1880143 // the
1880144 // socket: save the clock time, so
1880145 // that the
1880146 // same packet is not read again.
1880147 //
1880148 sfd->file->sock->read.clock[i]
1880149     = ip_table[i].clock;
1880150 //
1880151 // Copy the packet.
1880152 //
1880153 size_read
1880154     =
1880155     min (ntohs
1880156         (ip_table[i].packet.header.
1880157         tot_len), length);
1880158 //
1880159 memcpy (buffer,
1880160         ip_table[i].packet.octet,
1880161         size_read);
1880162 //
1880163 // Get the source address and
1880164 // return.
1880165 //
1880166 if (addrfrom != NULL && addrlen != NULL)
1880167     {
1880168         if (*addrlen >=
1880169             sizeof (struct sockaddr_in))
1880170             {
1880171                 addrfrom_in->sin_family = AF_INET;
1880172                 addrfrom_in->sin_port = 0;
1880173                 addrfrom_in->sin_addr.s_addr
1880174                     =
1880175                 ip_table[i].packet.header.saddr;
1880176             }
1880177         *addrlen =
1880178             sizeof (struct sockaddr_in);
1880179     }
```

```
1880180         return ((ssize_t) size_read);
1880181     }
1880182 }
1880183 else
1880184 {
1880185     //
1880186     // Unsupported protocol.
1880187     //
1880188     errset (EPROTONOSUPPORT);
1880189     return ((ssize_t) - 1);
1880190 }
1880191 }
1880192 else if (sfd->file->sock->type == SOCK_DGRAM)
1880193 {
1880194     //
1880195     // DGRAM
1880196     //
1880197     if (sfd->file->sock->protocol == IPPROTO_UDP)
1880198     {
1880199         //
1880200         // UDP
1880201         //
1880202         // Scan the ip_table[] to find an UDP
1880203         // packet
1880204         // that was not already seen by the
1880205         // socket.
1880206         //
1880207         for (i = 0; i < IP_MAX_PACKETS; i++)
1880208             {
1880209                 //
1880210                 // Check the protocol.
1880211                 //
1880212                 if (ip_table[i].packet.header.protocol !=
1880213                     IPPROTO_UDP)
1880214                     {
1880215                         //
1880216                         // It is not UDP.
```

```
1880217         //
1880218         continue;
1880219     }
1880220     //
1880221     // Is the packet new for the socket?
1880222     //
1880223     // Please notice that the kernel
1880224     // might be interrupted
1880225     // also between clock tics; so,
1880226     // during a single clock
1880227     // time, a new packet might be
1880228     // reached.
1880229     //
1880230     if (ip_table[i].clock
1880231         < sfd->file->sock->read.clock[i])
1880232     {
1880233         //
1880234         // Already seen or packet too
1880235         // old.
1880236         //
1880237         continue;
1880238     }
1880239     //
1880240     // Verify the ports.
1880241     //
1880242     udp = (struct udphdr *)
1880243         &ip_table[i].packet.octet
1880244         [sizeof (struct iphdr)];
1880245     //
1880246     if (udp->dest == 0)
1880247     {
1880248         //
1880249         // Cannot accept packets for the
1880250         // port zero!
1880251         //
1880252         continue;
1880253     }
```

```
1880254 //
1880255 if (udp->dest !=
1880256     htons (sfd->file->sock->lport))
1880257 {
1880258     //
1880259     // The local port does not
1880260     // match!
1880261     //
1880262     continue;
1880263 }
1880264 //
1880265 if (udp->source !=
1880266     htons (sfd->file->sock->rport)
1880267     && sfd->file->sock->rport != 0)
1880268 {
1880269     //
1880270     // The remote port does not
1880271     // match, and is not
1880272     // zero.
1880273     //
1880274     continue;
1880275 }
1880276 //
1880277 // Verify the IP addresses.
1880278 //
1880279 if (ip_table[i].packet.header.daddr
1880280     != htonl (sfd->file->sock->laddr)
1880281     && sfd->file->sock->laddr != 0)
1880282 {
1880283     //
1880284     // The local address does not
1880285     // match, and is
1880286     // not zero.
1880287     //
1880288     continue;
1880289 }
1880290 //
```



```
1880291     if (ip_table[i].packet.header.saddr
1880292         != htonl (sfd->file->sock->raddr)
1880293         && sfd->file->sock->raddr != 0)
1880294     {
1880295         //
1880296         // The remote address does not
1880297         // match, and is
1880298         // not zero.
1880299         //
1880300         continue;
1880301     }
1880302     //
1880303     // The packet is accepted.
1880304     //
1880305     // This UDP packet is new for the
1880306     // socket:
1880307     // save the clock time, so that the
1880308     // same packet is not read again.
1880309     //
1880310     sfd->file->sock->read.clock[i]
1880311         = ip_table[i].clock;
1880312     //
1880313     // Check the right minimal size to
1880314     // be read, comparing
1880315     // the size of the IP packet, the
1880316     // size of the UDP
1880317     // packet and the size requested.
1880318     //
1880319     size_read =
1880320         ntohs (ip_table[i].packet.header.
1880321             tot_len) -
1880322         (ip_table[i].packet.header.ihl * 4) -
1880323         (sizeof (struct udphdr));
1880324     size_read =
1880325         min (size_read,
1880326             (udp->len -
1880327                 sizeof (struct udphdr)));
```

```
1880328     size_read = min (size_read, length);
1880329     //
1880330     // Copy the data inside the UDP
1880331     // packet.
1880332     //
1880333     data =
1880334         ((uint8_t *) udp) +
1880335         sizeof (struct udphdr));
1880336     //
1880337     memcpy (buffer, data, size_read);
1880338     //
1880339     // Get the source address and
1880340     // return.
1880341     //
1880342     if (addrfrom != NULL && addrlen != NULL)
1880343     {
1880344         if (*addrlen >=
1880345             sizeof (struct sockaddr_in))
1880346         {
1880347             addrfrom_in->sin_family = AF_INET;
1880348             addrfrom_in->sin_port =
1880349                 udp->source;
1880350             addrfrom_in->sin_addr.s_addr =
1880351                 ip_table[i].packet.header.saddr;
1880352         }
1880353         *addrlen =
1880354             sizeof (struct sockaddr_in);
1880355     }
1880356     return ((ssize_t) size_read);
1880357 }
1880358 }
1880359 }
1880360 else if (sfd->file->sock->type == SOCK_STREAM)
1880361 {
1880362     //
1880363     // STREAM
1880364     //
```

```
1880365         if (sfd->file->sock->protocol == IPPROTO_TCP)
1880366             {
1880367                 //
1880368                 // TCP
1880369                 //
1880370                 // See if the read side of the stream
1880371                 // was closed.
1880372                 //
1880373                 if (sfd->file->sock->tcp.recv_closed
1880374                     || sfd->file->sock->tcp.conn == TCP_CLOSE)
1880375                     {
1880376                         //
1880377                         // If the 'recv_size' is zero, the
1880378                         // stream
1880379                         // is closed.
1880380                         //
1880381                         if (sfd->file->sock->tcp.recv_size
1880382                             == 0
1880383                             || sfd->file->sock->tcp.can_read == 0)
1880384                             {
1880385                                 //
1880386                                 // End of file.
1880387                                 //
1880388                                 return ((ssize_t) 0);
1880389                             }
1880390                     }
1880391                 //
1880392                 // At the moment, nothing was read.
1880393                 //
1880394                 size_read = 0;
1880395                 //
1880396                 // See if there is data to be read from
1880397                 // the stream.
1880398                 //
1880399                 if (sfd->file->sock->tcp.can_read)
1880400                     {
1880401                         size_read =
```

```
1880402         min (sfd->file->sock->tcp.recv_size,
1880403             length);
1880404     memcpy (buffer,
1880405            sfd->file->sock->tcp.recv_index,
1880406            size_read);
1880407     //
1880408     sfd->file->sock->tcp.recv_size -=
1880409         size_read;
1880410     sfd->file->sock->tcp.recv_index +=
1880411         size_read;
1880412     //
1880413     if (sfd->file->sock->tcp.recv_size == 0)
1880414     {
1880415         //
1880416         // Nothing to be read at the
1880417         // moment.
1880418         //
1880419         sfd->file->sock->tcp.can_read = 0;
1880420         sfd->file->sock->tcp.can_recv = 1;
1880421     }
1880422     //
1880423     // Get the source address and
1880424     // return.
1880425     //
1880426     if (addrfrom != NULL && addrlen != NULL)
1880427     {
1880428         if (*addrlen >=
1880429             sizeof (struct sockaddr_in))
1880430         {
1880431             addrfrom_in->sin_family = AF_INET;
1880432             addrfrom_in->sin_port =
1880433                 htons (sfd->file->sock->rport);
1880434             addrfrom_in->sin_addr.s_addr =
1880435                 htons (sfd->file->sock->raddr);
1880436         }
1880437         *addrlen =
1880438             sizeof (struct sockaddr_in);
```

```
1880439         }
1880440     }
1880441     //
1880442     // Check if something was read.
1880443     //
1880444     if (size_read > 0)
1880445     {
1880446         //
1880447         // Return normally.
1880448         //
1880449         return ((ssize_t) size_read);
1880450     }
1880451     else
1880452     {
1880453         //
1880454         // Nothing to be read at the moment.
1880455         //
1880456         if (sfd->fl_flags & O_NONBLOCK)
1880457         {
1880458             //
1880459             // Try again.
1880460             //
1880461             errset (EAGAIN);
1880462             return ((ssize_t) - 1);
1880463         }
1880464         else
1880465         {
1880466             //
1880467             // Go to sleep and return a
1880468             // temporary error.
1880469             //
1880470             proc_table[pid].status =
1880471                 PROC_SLEEPING;
1880472             proc_table[pid].ret = 0;
1880473             proc_table[pid].wakeup_events
1880474                 = WAKEUP_EVENT_SOCKET_READ;
1880475             proc_table[pid].wakeup_sock =
```

```
1880476         sfd->file->sock;
1880477     if (DEBUG)
1880478     {
1880479         k_printf
1880480         ("[%s:%i] PID %i goes to "
1880481          "sleep waiting to "
1880482          "receive for a socket.\n",
1880483          __FILE__, __LINE__, pid);
1880484     }
1880485     //
1880486     // Nothing was received.
1880487     //
1880488     errset (EAGAIN);
1880489     return ((ssize_t) - 1);
1880490 }
1880491 }
1880492 }
1880493 else
1880494 {
1880495     //
1880496     // Unsupported protocol.
1880497     //
1880498     errset (EPROTONOSUPPORT);
1880499     return ((ssize_t) - 1);
1880500 }
1880501 }
1880502 else
1880503 {
1880504     //
1880505     // Unsupported type.
1880506     //
1880507     errset (EPROTONOSUPPORT);
1880508     return ((ssize_t) - 1);
1880509 }
1880510 }
1880511 else
1880512 {
```

```
1880513     //
1880514     // Unsupported family.
1880515     //
1880516     errset (EAFNOSUPPORT);
1880517     return ((ssize_t) - 1);
1880518 }
1880519 //
1880520 // If we are here, there are no more packets to read
1880521 // at the moment.
1880522 //
1880523 if (sfd->fl_flags & O_NONBLOCK)
1880524 {
1880525     //
1880526     // Try again.
1880527     //
1880528     errset (EAGAIN);
1880529     return (-1);
1880530 }
1880531 else
1880532 {
1880533     //
1880534     // The process should go to sleep.
1880535     //
1880536     proc_table[pid].status = PROC_SLEEPING;
1880537     proc_table[pid].ret = 0;
1880538     proc_table[pid].wakeup_events =
1880539         WAKEUP_EVENT_SOCKET_READ;
1880540     proc_table[pid].wakeup_sock = sfd->file->sock;
1880541     if (DEBUG)
1880542     {
1880543         k_printf ("%s:%i] PID %i goes to sleep "
1880544                 "waiting to receive "
1880545                 "for a socket.\n",
1880546                 __FILE__, __LINE__, pid);
1880547     }
1880548     //
1880549     // Try again.
```

```
1880550         //
1880551         errset (EAGAIN);
1880552         return ((ssize_t) - 1);
1880553     }
1880554 }
```

94.8.31 kernel/lib_s/s_routeadd.c

«

Si veda la sezione [87.42](#).

```
1890001 #include <arpa/inet.h>
1890002 #include <sys/os32.h>
1890003 #include <kernel/net/route.h>
1890004 #include <kernel/lib_k.h>
1890005 #include <errno.h>
1890006 #include <netinet/in.h>
1890007 #include <kernel/proc.h>
1890008 //-----
1890009 // This syscall is present only inside os32.
1890010 //-----
1890011 int
1890012 s_routeadd (pid_t pid, in_addr_t dest, int m,
1890013             in_addr_t router, int device)
1890014 {
1890015     int r;
1890016     h_addr_t netmask;
1890017     h_addr_t network;
1890018     //
1890019     // Must be a privileged process.
1890020     //
1890021     if (proc_table[pid].euid != 0)
1890022     {
1890023         errset (EPERM);
1890024         return (-1);
1890025     }
1890026     //
1890027     //
```



```
1890028 //
1890029 if (m > 32 || m < 0)
1890030 {
1890031     errset (EINVAL);
1890032     return (-1);
1890033 }
1890034 //
1890035 // Calculate the netmask.
1890036 //
1890037 netmask = ip_mask (m);
1890038 //
1890039 // Fix the destination address, with the mask.
1890040 //
1890041 network = ntohl (dest) & netmask;
1890042 //
1890043 // Check if there is already. If there is: update
1890044 // it.
1890045 //
1890046 for (r = 0; r < ROUTE_MAX_ROUTES; r++)
1890047 {
1890048     if (network == route_table[r].network
1890049         && m == route_table[r].m)
1890050     {
1890051         //
1890052         // Update.
1890053         //
1890054         route_table[r].router = ntohl (router);
1890055         route_table[r].netmask = netmask;
1890056         route_table[r].interface = device;
1890057         return (0);
1890058     }
1890059 }
1890060 //
1890061 // The item is new. Find an empty place.
1890062 //
1890063 for (r = 0; r < ROUTE_MAX_ROUTES; r++)
1890064 {
```

```
1890065     if (route_table[r].network == 0xFFFFFFFF)
1890066     {
1890067         //
1890068         // Empty.
1890069         //
1890070         route_table[r].network = network;
1890071         route_table[r].netmask = netmask;
1890072         route_table[r].m = m;
1890073         route_table[r].router = ntohl (router);
1890074         route_table[r].interface = device;
1890075         //
1890076         route_sort ();
1890077         //
1890078         return (0);
1890079     }
1890080 }
1890081 //
1890082 // No free space found.
1890083 //
1890084 errset (ENOMEM);
1890085 return (-1);
1890086 }
```

94.8.32 kernel/lib_s/s_routedel.c



Si veda la sezione [87.43](#).

```
1900001 #include <arpa/inet.h>
1900002 #include <sys/os32.h>
1900003 #include <kernel/net/route.h>
1900004 #include <kernel/lib_k.h>
1900005 #include <errno.h>
1900006 #include <netinet/in.h>
1900007 #include <kernel/proc.h>
1900008 //-----
1900009 // This syscall is present only inside os32.
1900010 //-----
```

```
1900011 int
1900012 s_routedel (pid_t pid, in_addr_t dest, int m)
1900013 {
1900014     int r;
1900015     h_addr_t network;
1900016     //
1900017     // Must be a privileged process.
1900018     //
1900019     if (proc_table[pid].euid != 0)
1900020     {
1900021         errset (EPERM);
1900022         return (-1);
1900023     }
1900024     //
1900025     //
1900026     //
1900027     if (m > 32 || m < 0)
1900028     {
1900029         errset (EINVAL);
1900030         return (-1);
1900031     }
1900032     //
1900033     // Calculate the destination network with the mask.
1900034     //
1900035     network = ntohl (dest) & ip_mask (m);
1900036     //
1900037     // Check if there is already. If there is: remove
1900038     // it.
1900039     //
1900040     for (r = 0; r < ROUTE_MAX_ROUTES; r++)
1900041     {
1900042         if (network == route_table[r].network
1900043             && m == route_table[r].m)
1900044         {
1900045             //
1900046             // Remove.
1900047             //
```

```
1900048         memset (&route_table[m], 0xFF,
1900049                 sizeof (route_table[m]));
1900050         return (0);
1900051     }
1900052 }
1900053 //
1900054 // Not found.
1900055 //
1900056 errset (EINVAL);
1900057 return (-1);
1900058 }
```

94.8.33 kernel/lib_s/s_sbrk.c

<<

Si veda la sezione [87.5](#).

```
1910001 #include <errno.h>
1910002 #include <kernel/proc.h>
1910003 #include <kernel/lib_k.h>
1910004 #include <kernel/lib_s.h>
1910005 //-----
1910006 void *
1910007 s_sbrk (pid_t pid, intptr_t increment)
1910008 {
1910009     size_t previous_size;
1910010     size_t new_size;
1910011     int status;
1910012     //
1910013     // Get current data segment full size.
1910014     //
1910015     if (proc_table[pid].domain_data == 0)
1910016     {
1910017         previous_size = (proc_table[pid].domain_text
1910018                         + proc_table[pid].extra_data);
1910019     }
1910020     else
1910021     {
```

```
1910022     previous_size = (proc_table[pid].domain_data
1910023                 + proc_table[pid].extra_data);
1910024     }
1910025     //
1910026     // Check increment.
1910027     //
1910028     if ((increment + proc_table[pid].extra_data) < 0)
1910029     {
1910030         //
1910031         // Cannot reduce too much. Just correct it.
1910032         //
1910033         increment = -proc_table[pid].extra_data;
1910034     }
1910035     //
1910036     // Calculate the new size.
1910037     //
1910038     new_size = previous_size + increment;
1910039     //
1910040     // Call 's_brk()' to do the work. The new size value
1910041     // is the
1910042     // same of the new requested pointer address.
1910043     //
1910044     status = s_brk (pid, (void *) new_size);
1910045     //
1910046     if (status < 0)
1910047     {
1910048         errset (errno);
1910049         return ((void *) -1);
1910050     }
1910051     //
1910052     // Ok: return previous final address.
1910053     //
1910054     return ((void *) previous_size);
1910055 }
```

94.8.34 kernel/lib_s/s_send.c



Si veda la sezione [87.45](#).

```
1920001 #include <kernel/proc.h>
1920002 #include <kernel/net.h>
1920003 #include <kernel/net/route.h>
1920004 #include <kernel/net/udp.h>
1920005 #include <errno.h>
1920006 #include <arpa/inet.h>
1920007 #include <fcntl.h>
1920008 #include <sys/os32.h>
1920009 //-----
1920010 #define DEBUG 0
1920011 //-----
1920012 ssize_t
1920013 s_send (pid_t pid, int sfdn, const void *buffer,
1920014         size_t size, int flags)
1920015 {
1920016     fd_t *sfd;
1920017     int status;
1920018     //
1920019     // Get file descriptor and verify that it is a
1920020     // socket.
1920021     //
1920022     sfd = fd_reference (pid, &sfdn);
1920023     if (sfd == NULL || sfd->file == NULL)
1920024     {
1920025         errset (EBADF);    // Bad file descriptor.
1920026         return ((ssize_t) - 1);
1920027     }
1920028     if (sfd->file->sock == NULL)
1920029     {
1920030         errset (ENOTSOCK);    // Not a socket.
1920031         return ((ssize_t) - 1);
1920032     }
1920033     if (sfd->file->sock->unreach_port)
1920034     {
```

```
1920035     errset (ECONNREFUSED);    // Connection refused.
1920036     return ((ssize_t) - 1);
1920037 }
1920038 if (sfd->file->sock->unreach_prot)
1920039 {
1920040     errset (ENOPROTOOPT);    // Protocol not
1920041     // available.
1920042     return ((ssize_t) - 1);
1920043 }
1920044 if (sfd->file->sock->unreach_host)
1920045 {
1920046     errset (EHOSTUNREACH);  // Host unreachable.
1920047     return ((ssize_t) - 1);
1920048 }
1920049 if (sfd->file->sock->unreach_net)
1920050 {
1920051     errset (ENETUNREACH);   // Net unreachable.
1920052     return ((ssize_t) - 1);
1920053 }
1920054 //
1920055 // Verify to have a valid buffer pointer.
1920056 //
1920057 if (buffer == NULL)
1920058 {
1920059     errset (EINVAL);
1920060     return ((ssize_t) - 1);
1920061 }
1920062 //
1920063 //
1920064 //
1920065 if (sfd->file->sock->family == AF_INET)
1920066 {
1920067     //
1920068     // INET
1920069     //
1920070     // AF_INET requires at least the remote address.
1920071     //
```

```
1920072     if (sfd->file->sock->raddr == 0)
1920073     {
1920074         errset (EDESTADDRREQ);
1920075         return ((ssize_t) - 1);
1920076     }
1920077     //
1920078     if (sfd->file->sock->type == SOCK_RAW)
1920079     {
1920080         //
1920081         // RAW
1920082         //
1920083         if (sfd->file->sock->protocol == IPPROTO_ICMP)
1920084         {
1920085             //
1920086             // ICMP
1920087             //
1920088             status = ip_tx (sfd->file->sock->laddr,
1920089                            sfd->file->sock->raddr,
1920090                            sfd->file->sock->protocol,
1920091                            buffer, size);
1920092             if (status)
1920093             {
1920094                 errset (errno);
1920095                 return ((ssize_t) - 1);
1920096             }
1920097             else
1920098             {
1920099                 return ((ssize_t) size);
1920100             }
1920101         }
1920102     else
1920103     {
1920104         //
1920105         // Unsupported protocol.
1920106         //
1920107         errset (EPROTONOSUPPORT);
1920108         return ((ssize_t) - 1);
```



```
1920109     }
1920110 }
1920111 else if (sfd->file->sock->type == SOCK_DGRAM)
1920112 {
1920113     //
1920114     // DGRAM
1920115     //
1920116     if (sfd->file->sock->protocol == IPPROTO_UDP)
1920117     {
1920118         //
1920119         // UDP
1920120         //
1920121         status = udp_tx (sfd->file->sock->lport,
1920122                         sfd->file->sock->rport,
1920123                         sfd->file->sock->laddr,
1920124                         sfd->file->sock->raddr,
1920125                         buffer, size);
1920126
1920127         if (status)
1920128         {
1920129             errset (errno);
1920130             return ((ssize_t) - 1);
1920131         }
1920132     else
1920133     {
1920134         return ((ssize_t) size);
1920135     }
1920136 else
1920137 {
1920138     //
1920139     // Unsupported protocol.
1920140     //
1920141     errset (EPROTONOSUPPORT);
1920142     return ((ssize_t) - 1);
1920143 }
1920144 }
1920145 else if (sfd->file->sock->type == SOCK_STREAM)
```

```
1920146     {
1920147         //
1920148         // STREAM
1920149         //
1920150         if (sfd->file->sock->protocol == IPPROTO_TCP)
1920151         {
1920152             //
1920153             // TCP
1920154             //
1920155             // See if the send side of the stream
1920156             // was closed.
1920157             //
1920158             if (sfd->file->sock->tcp.send_closed
1920159                 || sfd->file->sock->tcp.conn == TCP_CLOSE)
1920160             {
1920161                 //
1920162                 // End of file.
1920163                 //
1920164                 if (DEBUG)
1920165                 {
1920166                     k_printf ("end of socket write\n");
1920167                 }
1920168                 s_kill ((pid_t) 0, pid, SIGPIPE);
1920169                 errset (EPIPE);
1920170                 return ((ssize_t) - 1);
1920171             }
1920172             //
1920173             // Put data to the send buffer, if it is
1920174             // possible.
1920175             //
1920176             if (sfd->file->sock->tcp.can_write)
1920177             {
1920178                 size =
1920179                     min (size,
1920180                         (TCP_MSS -
1920181                          sizeof (struct tcphdr)));
1920182                 memcpy (sfd->file->sock->tcp.send_data,
```

```
1920183         buffer, size);
1920184     sfd->file->sock->tcp.send_size = size;
1920185     sfd->file->sock->tcp.can_write = 0;
1920186     sfd->file->sock->tcp.can_send = 1;
1920187     //
1920188     sfd->file->sock->tcp.lsq[++sfd->
1920189         file->sock->tcp.
1920190         lsqi] =
1920191         sfd->file->sock->tcp.lsq_ack;
1920192     sfd->file->sock->tcp.send_flags =
1920193         TCP_FLAG_PSH | TCP_FLAG_ACK;
1920194     tcp_tx_sock (sfd->file->sock);
1920195     //
1920196     return ((ssize_t) size);
1920197 }
1920198 else
1920199 {
1920200     //
1920201     // At the moment, nothing can be
1920202     // written.
1920203     //
1920204     if (sfd->fl_flags & O_NONBLOCK)
1920205     {
1920206         //
1920207         // Cannot block.
1920208         //
1920209         errset (EAGAIN);
1920210         return ((ssize_t) - 1);
1920211     }
1920212     else
1920213     {
1920214         //
1920215         // Go to sleep and return zero.
1920216         //
1920217         proc_table[pid].status =
1920218             PROC_SLEEPING;
1920219         proc_table[pid].ret = 0;
```

```
1920220         proc_table[pid].wakeup_events
1920221             = WAKEUP_EVENT_SOCKET_WRITE;
1920222         proc_table[pid].wakeup_sock =
1920223             sfd->file->sock;
1920224         if (DEBUG)
1920225             {
1920226                 k_printf
1920227                     ("[%s] PID %i goes to "
1920228                      "sleep waiting to write "
1920229                      "to a socket.\n",
1920230                     __FILE__, pid);
1920231             }
1920232         //
1920233         // Retry.
1920234         //
1920235         errset (EAGAIN);
1920236         return ((ssize_t) - 1);
1920237     }
1920238 }
1920239 }
1920240 else
1920241 {
1920242     //
1920243     // Unsupported protocol.
1920244     //
1920245     errset (EPROTONOSUPPORT);
1920246     return ((ssize_t) - 1);
1920247 }
1920248 }
1920249 else
1920250 {
1920251     //
1920252     // Unsupported type.
1920253     //
1920254     errset (EPROTONOSUPPORT);
1920255     return ((ssize_t) - 1);
1920256 }
```

```
1920257     }
1920258     else
1920259     {
1920260         //
1920261         // Unsupported family.
1920262         //
1920263         errset (EAFNOSUPPORT);
1920264         return ((ssize_t) - 1);
1920265     }
1920266 }
```

94.8.35 kernel/lib_s/s_setegid.c

Si veda la sezione [87.48](#).



```
1930001 #include <kernel/proc.h>
1930002 #include <kernel/lib_s.h>
1930003 #include <errno.h>
1930004 //-----
1930005 int
1930006 s_setegid (pid_t pid, gid_t egid)
1930007 {
1930008     if ((proc_table[pid].euid == 0)
1930009         || (proc_table[pid].egid == 0))
1930010     {
1930011         proc_table[pid].egid = egid;
1930012         return (0);
1930013     }
1930014     else if (egid == proc_table[pid].egid)
1930015     {
1930016         return (0);
1930017     }
1930018     else if (egid == proc_table[pid].gid
1930019             || egid == proc_table[pid].sgid)
1930020     {
1930021         proc_table[pid].egid = egid;
1930022         return (0);
```

```
1930023     }
1930024     else
1930025     {
1930026         errset (EPERM);
1930027         return (-1);
1930028     }
1930029 }
```

94.8.36 kernel/lib_s/s_seteuid.c

«

Si veda la sezione [87.51](#).

```
1940001 #include <kernel/proc.h>
1940002 #include <kernel/lib_s.h>
1940003 #include <errno.h>
1940004 //-----
1940005 int
1940006 s_seteuid (pid_t pid, uid_t euid)
1940007 {
1940008     if (proc_table[pid].euid == 0)
1940009     {
1940010         proc_table[pid].euid = euid;
1940011         return (0);
1940012     }
1940013     else if (euid == proc_table[pid].euid)
1940014     {
1940015         return (0);
1940016     }
1940017     else if (euid == proc_table[pid].uid
1940018             || euid == proc_table[pid].suid)
1940019     {
1940020         proc_table[pid].euid = euid;
1940021         return (0);
1940022     }
1940023     else
1940024     {
1940025         errset (EPERM);
```

```
1940026     return (-1);
1940027     }
1940028 }
```

94.8.37 kernel/lib_s/s_setgid.c

Si veda la sezione [87.48](#).

```
1950001 #include <kernel/proc.h>
1950002 #include <kernel/lib_s.h>
1950003 #include <errno.h>
1950004 //-----
1950005 int
1950006 s_setgid (pid_t pid, gid_t gid)
1950007 {
1950008     if ((proc_table[pid].euid == 0)
1950009         || (proc_table[pid].egid == 0))
1950010     {
1950011         proc_table[pid].gid = gid;
1950012         proc_table[pid].egid = gid;
1950013         proc_table[pid].sgid = gid;
1950014         return (0);
1950015     }
1950016     else if (gid == proc_table[pid].egid)
1950017     {
1950018         return (0);
1950019     }
1950020     else if (gid == proc_table[pid].gid
1950021             || gid == proc_table[pid].sgid)
1950022     {
1950023         proc_table[pid].egid = gid;
1950024         return (0);
1950025     }
1950026     else
1950027     {
1950028         errset (EPERM);
1950029         return (-1);
```

1950030	}
1950031	}

94.8.38 kernel/lib_s/s_setjmp.c

<<

Si veda la sezione [87.49](#).

```
1960001 #include <kernel/lib_s.h>
1960002 #include <kernel/proc.h>
1960003 #include <errno.h>
1960004 #include <setjmp.h>
1960005 //-----
1960006 extern uint32_t proc_stack_pointer;
1960007 //-----
1960008 int
1960009 s_setjmp (pid_t pid, jmp_buf env)
1960010 {
1960011     jmp_stack_t *sp;
1960012     jmp_env_t *jmpenv;
1960013     //
1960014     // Find where is the process stack in memory, from
1960015     // the kernel point
1960016     // of view. Please notice that the current stack at
1960017     // 'proc_stack_pointer' will be saved from the
1960018     // scheduler inside
1960019     // the process table, and current stack saved inside
1960020     // the process
1960021     // table is not up to date.
1960022     //
1960023     sp = ptr (pid, (void *) proc_stack_pointer);
1960024     //
1960025     // Translate the pointer 'env', to the kernel point
1960026     // of view.
1960027     //
1960028     jmpenv = ptr (pid, env);
1960029     //
1960030     // Save the process stack.
```



```
1960031 //
1960032 jmpenv->eax0 = sp->eax0;
1960033 jmpenv->ecx0 = sp->ecx0;
1960034 jmpenv->edx0 = sp->edx0;
1960035 jmpenv->ebx0 = sp->ebx0;
1960036 jmpenv->ebp0 = sp->ebp0;
1960037 jmpenv->esi0 = sp->esi0;
1960038 jmpenv->edi0 = sp->edi0;
1960039 jmpenv->ds0 = sp->ds0;
1960040 jmpenv->es0 = sp->es0;
1960041 jmpenv->fs0 = sp->fs0;
1960042 jmpenv->gs0 = sp->gs0;
1960043 jmpenv->eflags0 = sp->eflags0;
1960044 jmpenv->cs0 = sp->cs0;
1960045 jmpenv->eip0 = sp->eip0;
1960046 //
1960047 jmpenv->eip1 = sp->eip1;
1960048 jmpenv->syscallnr = sp->syscallnr;
1960049 jmpenv->msg_pointer = sp->msg_pointer;
1960050 jmpenv->msg_size = sp->msg_size;
1960051 jmpenv->env = sp->env;
1960052 jmpenv->ret = sp->ret;
1960053 jmpenv->ebp1 = sp->ebp1;
1960054 jmpenv->eip2 = sp->eip2;
1960055 //
1960056 // Save also the stack pointer!
1960057 //
1960058 jmpenv->esp0 = proc_stack_pointer;
1960059 //
1960060 return 0;
1960061 }
```

94.8.39 kernel/lib_s/s_setuid.c



Si veda la sezione [87.51](#).

```
1970001 #include <kernel/proc.h>
1970002 #include <kernel/lib_s.h>
1970003 #include <errno.h>
1970004 //-----
1970005 int
1970006 s_setuid (pid_t pid, uid_t uid)
1970007 {
1970008     if (proc_table[pid].euid == 0)
1970009     {
1970010         proc_table[pid].uid = uid;
1970011         proc_table[pid].euid = uid;
1970012         proc_table[pid].suid = uid;
1970013         return (0);
1970014     }
1970015     else if (uid == proc_table[pid].euid)
1970016     {
1970017         return (0);
1970018     }
1970019     else if (uid == proc_table[pid].uid
1970020             || uid == proc_table[pid].suid)
1970021     {
1970022         proc_table[pid].euid = uid;
1970023         return (0);
1970024     }
1970025     else
1970026     {
1970027         errset (EPERM);
1970028         return (-1);
1970029     }
1970030 }
```

94.8.40 kernel/lib_s/s_signal.c



Si veda la sezione [87.52](#).

```
1980001 #include <kernel/lib_s.h>
1980002 #include <kernel/proc.h>
1980003 #include <errno.h>
1980004 //-----
1980005 sighandler_t
1980006 s_signal (pid_t pid, int sig, sighandler_t handler,
1980007          uintptr_t wrapper)
1980008 {
1980009     unsigned long int flag = 1L << (sig - 1);
1980010     sighandler_t previous;
1980011     //
1980012     if (sig <= 0)
1980013     {
1980014         errset (EINVAL);
1980015         return (SIG_ERR);
1980016     }
1980017     //
1980018     if (proc_table[pid].sig_ignore & flag)
1980019     {
1980020         previous = SIG_IGN;
1980021     }
1980022     else if (proc_table[pid].sig_handler[sig] !=
1980023             (uintptr_t) NULL)
1980024     {
1980025         previous =
1980026             (sighandler_t) proc_table[pid].sig_handler[sig];
1980027     }
1980028     else
1980029     {
1980030         previous = SIG_DFL;
1980031     }
1980032     //
1980033     if (handler == SIG_DFL)
1980034     {
```

```
1980035     //
1980036     // Enable signal.
1980037     //
1980038     proc_table[pid].sig_ignore &= ~flag;
1980039     //
1980040     return (previous);
1980041 }
1980042 else if (handler == SIG_IGN)
1980043 {
1980044     //
1980045     // Disable signal.
1980046     //
1980047     proc_table[pid].sig_ignore |= flag;
1980048     //
1980049     return (previous);
1980050 }
1980051 else
1980052 {
1980053     //
1980054     // Enable signal, store the handler address and
1980055     // the
1980056     // handler-wrapper.
1980057     //
1980058     proc_table[pid].sig_ignore &= ~flag;
1980059     proc_table[pid].sig_handler[sig] =
1980060         (uintptr_t) handler;
1980061     proc_table[pid].sig_handler_wrapper = wrapper;
1980062     //
1980063     return (previous);
1980064 }
1980065 }
```

94.8.41 kernel/lib_s/s_socket.c



Si veda la sezione [87.54](#).

```
1990001 #include <kernel/proc.h>
1990002 #include <kernel/lib_s.h>
1990003 #include <kernel/lib_k.h>
1990004 #include <errno.h>
1990005 #include <fcntl.h>
1990006 #include <sys/socket.h>
1990007 #include <arpa/inet.h>
1990008 //-----
1990009 int
1990010 s_socket (pid_t pid, int family, int type, int protocol)
1990011 {
1990012     fd_t *fd;
1990013     int sfdn;
1990014     file_t *file;
1990015     sock_t *sock;
1990016     //
1990017     // Check supported family type.
1990018     //
1990019     if (family != AF_INET)
1990020     {
1990021         errset (EAFNOSUPPORT);
1990022         return (-1);
1990023     }
1990024     //
1990025     // Check supported communication type.
1990026     //
1990027     if (type == SOCK_RAW || type == SOCK_DGRAM
1990028         || type == SOCK_STREAM)
1990029     {
1990030         //
1990031         // Ok.
1990032         //
1990033         ;
1990034     }
```

```
1990035     else
1990036     {
1990037         errset (EPROTONOSUPPORT);
1990038         return (-1);
1990039     }
1990040     //
1990041     // Check supported protocol type.
1990042     //
1990043     if (protocol == IPPROTO_ICMP
1990044         || protocol == IPPROTO_UDP || protocol == IPPROTO_TCP)
1990045     {
1990046         //
1990047         // Ok.
1990048         //
1990049         ;
1990050     }
1990051     else
1990052     {
1990053         errset (EPROTONOSUPPORT);
1990054         return (-1);
1990055     }
1990056     //
1990057     // If it is a raw socket, must be a privileged
1990058     // process.
1990059     //
1990060     if (type == SOCK_RAW && proc_table[pid].euid != 0)
1990061     {
1990062         errset (EACCES);
1990063         return (-1);
1990064     }
1990065     //
1990066     // Find a free slot inside the sock_table[].
1990067     //
1990068     sock = sock_reference (-1);
1990069     if (sock == NULL)
1990070     {
1990071         errset (ENFILE);
```

```
1990072     return (-1);
1990073     }
1990074     //
1990075     // Find a free slot inside the file table.
1990076     //
1990077     file = file_reference (-1);
1990078     if (file == NULL)
1990079     {
1990080         errset (ENFILE); // Too many files open in
1990081         // system.
1990082         return (-1);
1990083     }
1990084     //
1990085     // Find a free slot inside the file descriptor
1990086     // table.
1990087     // Variable 'sfdn' will be modified by the call to
1990088     // 'fd_reference()'.
1990089     //
1990090     sfdn = -1;
1990091     fd = fd_reference (pid, &sfdn);
1990092     if (fd == NULL)
1990093     {
1990094         errset (EMFILE); // Too many open files.
1990095         return (-1);
1990096     }
1990097     //
1990098     // socket, system file and file descriptor ready:
1990099     // reset and put data
1990100     // inside them. Please notice that the
1990101     // tcp.listen_queue[] array is
1990102     // reset with all 0xFF, because zero is a valid file
1990103     // descriptor
1990104     // number.
1990105     //
1990106     memset (sock, 0, sizeof (sock_t));
1990107     sock->active = 1;
1990108     sock->family = family;
```

```
1990109     sock->type = type;
1990110     sock->protocol = protocol;
1990111     sock->lport = 0;
1990112     sock->laddr = 0;
1990113     sock->rport = 0;
1990114     sock->raddr = 0;
1990115     memset (sock->read.clock, 0x00,
1990116             sizeof (sock->read.clock));
1990117     memset (sock->tcp.listen_queue, 0xFF,
1990118             sizeof (sock->tcp.listen_queue));
1990119     //
1990120     file->references = 1;
1990121     file->oflags = O_RDWR;
1990122     file->inode = NULL;
1990123     file->sock = sock;
1990124     file->offset = 0;
1990125     //
1990126     fd->fl_flags = (O_RDWR | O_APPEND);
1990127     fd->fd_flags = 0;
1990128     fd->file = file;
1990129     //
1990130     // Return the file descriptor.
1990131     //
1990132     return (sfdn);
1990133 }
```

94.8.42 kernel/lib_s/s_stat.c



Si veda la sezione [87.55](#).

```
2000001     #include <kernel/fs.h>
2000002     #include <errno.h>
2000003     #include <kernel/proc.h>
2000004     #include <kernel/lib_s.h>
2000005     //-----
2000006     int
2000007     s_stat (pid_t pid, const char *path, struct stat *buffer)
```



```
2000008 {
2000009     proc_t *ps;
2000010     inode_t *inode;
2000011     //
2000012     // Check path.
2000013     //
2000014     if (path == NULL || strlen (path) == 0)
2000015     {
2000016         errset (EINVAL);
2000017         return (-1);
2000018     }
2000019     //
2000020     // Get process.
2000021     //
2000022     ps = proc_reference (pid);
2000023     //
2000024     // Try to load the file inode.
2000025     //
2000026     inode = path_inode (pid, path);
2000027     if (inode == NULL)
2000028     {
2000029         //
2000030         // Cannot access the file: it does not exists or
2000031         // permissions are
2000032         // not sufficient. Variable 'errno' is set by
2000033         // function
2000034         // 'path_inode()'.
2000035         //
2000036         errset (errno);
2000037         return (-1);
2000038     }
2000039     //
2000040     // Inode loaded: update the buffer.
2000041     //
2000042     buffer->st_dev = inode->sb->device;
2000043     buffer->st_ino = inode->ino;
2000044     buffer->st_mode = inode->mode;
```

```
2000045     buffer->st_nlink = inode->nlink;
2000046     buffer->st_uid = inode->uid;
2000047     buffer->st_gid = inode->gid;
2000048     if (S_ISBLK (buffer->st_mode)
2000049         || S_ISCHR (buffer->st_mode))
2000050     {
2000051         buffer->st_rdev = inode->direct[0];
2000052     }
2000053     else
2000054     {
2000055         buffer->st_rdev = 0;
2000056     }
2000057     buffer->st_size = inode->size;
2000058     buffer->st_atime = inode->time;           // All times
2000059     // are the
2000060     // same for
2000061     buffer->st_mtime = inode->time;         // Minix 1
2000062     // file
2000063     // system.
2000064     buffer->st_ctime = inode->time;         //
2000065     buffer->st_blksize = inode->sb->blksize;
2000066     buffer->st_blocks = inode->blkcnt;
2000067     //
2000068     // If the inode is a device special file, the
2000069     // 'st_rdev' value is
2000070     // taken from the first direct zone (as of Minix 1
2000071     // organization).
2000072     //
2000073     if (S_ISBLK (inode->mode) || S_ISCHR (inode->mode))
2000074     {
2000075         buffer->st_rdev = inode->direct[0];
2000076     }
2000077     else
2000078     {
2000079         buffer->st_rdev = 0;
2000080     }
2000081     //
```

```
2000082 // Release the inode and return.
2000083 //
2000084 inode_put (inode);
2000085 //
2000086 // Return.
2000087 //
2000088 return (0);
2000089 }
```

94.8.43 kernel/lib_s/s_stime.c



Si veda la sezione [87.59](#).

```
2010001 #include <kernel/lib_s.h>
2010002 #include <kernel/proc.h>
2010003 #include <stddef.h>
2010004 #include <errno.h>
2010005 //-----
2010006 extern clock_t _clock_time; // uint64_t
2010007 //-----
2010008 int
2010009 s_stime (pid_t pid, time_t * timer)
2010010 {
2010011     if (proc_table[pid].euid != 0)
2010012     {
2010013         errset (EPERM);
2010014         return (-1);
2010015     }
2010016 //
2010017     _clock_time = *timer * CLOCKS_PER_SEC;
2010018     return (0);
2010019 }
```

94.8.44 kernel/lib_s/s_tcgetattr.c

<<

Si veda la sezione [87.58](#).

```
2020001 #include <kernel/fs.h>
2020002 #include <errno.h>
2020003 #include <kernel/proc.h>
2020004 #include <kernel/lib_s.h>
2020005 #include <termios.h>
2020006 #include <sys/types.h>
2020007 //-----
2020008 int
2020009 s_tcgetattr (pid_t pid, int fdn, struct termios *termios_p)
2020010 {
2020011     file_t *file;
2020012     inode_t *inode;
2020013     dev_t device;
2020014     int t;           // 'tty_table[]' subscript.
2020015     int c;           // 'c_cc[]' subscript.
2020016     //
2020017     file = proc_table[pid].fd[fdn].file;
2020018     //
2020019     if (file == NULL)
2020020     {
2020021         errset (EBADF);
2020022         return (-1);
2020023     }
2020024     //
2020025     inode = proc_table[pid].fd[fdn].file->inode;
2020026     //
2020027     if (inode == NULL)
2020028     {
2020029         errset (EBADF);
2020030         return (-1);
2020031     }
2020032     //
2020033     if (!S_ISCHR (inode->mode))
2020034     {
```

```
2020035     errset (ENOTTY);
2020036     return (-1);
2020037 }
2020038 //
2020039 device = inode->direct[0];
2020040 //
2020041 if (major (device) != DEV_CONSOLE_MAJOR)
2020042 {
2020043     errset (ENOTTY);
2020044     return (-1);
2020045 }
2020046 //
2020047 t = minor (device);
2020048 if (t >= TTYS_TOTAL)
2020049 {
2020050     errset (ENOTTY);
2020051     return (-1);
2020052 }
2020053 //
2020054 // Ok: copy data.
2020055 //
2020056 termios_p->c_iflag = tty_table[t].attr.c_iflag;
2020057 termios_p->c_oflag = tty_table[t].attr.c_oflag;
2020058 termios_p->c_cflag = tty_table[t].attr.c_cflag;
2020059 termios_p->c_lflag = tty_table[t].attr.c_lflag;
2020060 for (c = 0; c < NCCS; c++)
2020061 {
2020062     termios_p->c_cc[c] = tty_table[t].attr.c_cc[c];
2020063 }
2020064 //
2020065 // Ok.
2020066 //
2020067 return (0);
2020068 }
```

94.8.45 kernel/lib_s/s_tcsetattr.c



Si veda la sezione [87.58](#).

```
2030001 #include <kernel/fs.h>
2030002 #include <errno.h>
2030003 #include <kernel/proc.h>
2030004 #include <kernel/lib_s.h>
2030005 #include <termios.h>
2030006 #include <sys/types.h>
2030007 //-----
2030008 // The following are masks of the implemented
2030009 // attributes.
2030010 //
2030011 #define MASK_C_IFLAG (BRKINT|ICRNL|IGNBRK|IGNCR)
2030012 #define MASK_C_OFLAG 0
2030013 #define MASK_C_CFLAG 0
2030014 #define MASK_C_LFLAG \
2030015     (ECHO|ECHOE|ECHOK|ECHONL|ICANON|ISIG)
2030016 //-----
2030017 int
2030018 s_tcsetattr (pid_t pid, int fdn, int action,
2030019             struct termios *termios_p)
2030020 {
2030021     file_t *file;
2030022     inode_t *inode;
2030023     dev_t device;
2030024     int t;           // 'tty_table[]' subscript.
2030025     int c;           // 'c_cc[]' subscript.
2030026     //
2030027     file = proc_table[pid].fd[fdn].file;
2030028     //
2030029     if (file == NULL)
2030030     {
2030031         errset (EBADF);
2030032         return (-1);
2030033     }
2030034     //
```

```
2030035     inode = proc_table[pid].fd[fdn].file->inode;
2030036     //
2030037     if (inode == NULL)
2030038     {
2030039         errset (EBADF);
2030040         return (-1);
2030041     }
2030042     //
2030043     if (!S_ISCHR (inode->mode))
2030044     {
2030045         errset (ENOTTY);
2030046         return (-1);
2030047     }
2030048     //
2030049     device = inode->direct[0];
2030050     //
2030051     if (major (device) != DEV_CONSOLE_MAJOR)
2030052     {
2030053         errset (ENOTTY);
2030054         return (-1);
2030055     }
2030056     //
2030057     t = minor (device);
2030058     if (t >= TTYS_TOTAL)
2030059     {
2030060         errset (ENOTTY);
2030061         return (-1);
2030062     }
2030063     //
2030064     // The parameter 'actions' is silently ignored: only
2030065     // immediate update will take place.
2030066     //
2030067     // The function will not notice if at least a
2030068     // successful attribute change is done, so,
2030069     // after this point, the return value is
2030070     // always zero.
2030071     //
```

```
2030072 tty_table[t].attr.c_iflag =
2030073     (termios_p->c_iflag & MASK_C_IFLAG);
2030074 tty_table[t].attr.c_oflag =
2030075     (termios_p->c_oflag & MASK_C_OFLAG);
2030076 tty_table[t].attr.c_cflag =
2030077     (termios_p->c_cflag & MASK_C_CFLAG);
2030078 tty_table[t].attr.c_lflag =
2030079     (termios_p->c_lflag & MASK_C_LFLAG);
2030080 for (c = 0; c < NCCS; c++)
2030081     {
2030082         //
2030083         // Should be done some check here?
2030084         //
2030085         tty_table[t].attr.c_cc[c] = termios_p->c_cc[c];
2030086     }
2030087     //
2030088     // Ok.
2030089     //
2030090     return (0);
2030091 }
```

94.8.46 kernel/lib_s/s_time.c



Si veda la sezione [87.59](#).

```
2040001 #include <kernel/lib_k.h>
2040002 #include <kernel/lib_s.h>
2040003 #include <stddef.h>
2040004 //-----
2040005 extern clock_t _clock_time;      // uint64_t
2040006 //-----
2040007 time_t
2040008 s_time (pid_t pid, time_t * timer)
2040009 {
2040010     time_t time = _clock_time / CLOCKS_PER_SEC;
2040011     if (timer != NULL)
2040012     {
```



```
2040013     *timer = time;
2040014     }
2040015     return (time);
2040016 }
```

94.8.47 kernel/lib_s/s_umount.c

Si veda la sezione [87.36](#).

```
2050001 #include <kernel/fs.h>
2050002 #include <errno.h>
2050003 #include <kernel/proc.h>
2050004 #include <kernel/lib_s.h>
2050005 //-----
2050006 int
2050007 s_umount (pid_t pid, const char *path_mnt)
2050008 {
2050009     proc_t *ps;
2050010     dev_t device; // Device to mount.
2050011     inode_t *inode_mount_point; // Original mount
2050012     // point.
2050013     inode_t *inode; // Inode table.
2050014     int i; // Inode table index.
2050015     //
2050016     // Get process.
2050017     //
2050018     ps = proc_reference (pid);
2050019     //
2050020     // Verify to be the super user.
2050021     //
2050022     if (ps->euid != 0)
2050023     {
2050024         errset (EPERM); // Operation not permitted.
2050025         return (-1);
2050026     }
2050027     //
2050028     // Get the directory mount point.
```

```
2050029 //
2050030 inode_mount_point = path_inode (pid, path_mnt);
2050031 if (inode_mount_point == NULL)
2050032 {
2050033     errset (ENOENT); // No such file or directory.
2050034     return (-1);
2050035 }
2050036 //
2050037 // Verify that the path is a directory.
2050038 //
2050039 if (!S_ISDIR (inode_mount_point->mode))
2050040 {
2050041     inode_put (inode_mount_point);
2050042     errset (ENOTDIR); // Not a directory.
2050043     return (-1);
2050044 }
2050045 //
2050046 // Verify that there is something attached.
2050047 //
2050048 device = inode_mount_point->sb_attached->device;
2050049 if (device == 0)
2050050 {
2050051     //
2050052     // There is nothing to unmount.
2050053     //
2050054     inode_put (inode_mount_point);
2050055     errset (E_NOT_MOUNTED); // Not mounted.
2050056     return (-1);
2050057 }
2050058 //
2050059 // Are there exactly two internal references? Let's
2050060 // explain:
2050061 // the directory that act as mount point, should
2050062 // have one reference
2050063 // because it is mounting something and another
2050064 // because it was just
2050065 // opened again, a few lines above. If there are
```

```
2050066 // more references
2050067 // it is wrong; if there are less, it is also wrong
2050068 // at this point.
2050069 //
2050070 if (inode_mount_point->references != 2)
2050071 {
2050072     inode_put (inode_mount_point);
2050073     errset (EUNKNOWN); // Unknown error.
2050074     return (-1);
2050075 }
2050076 //
2050077 // All data is available: find if there are open
2050078 // file inside
2050079 // the file system to unmount. But first load the
2050080 // inode table
2050081 // pointer.
2050082 //
2050083 inode = inode_reference ((dev_t) 0, (ino_t) 0);
2050084 if (inode == NULL)
2050085 {
2050086     //
2050087     // This error should not happen.
2050088     //
2050089     inode_put (inode_mount_point);
2050090     errset (EUNKNOWN); // Unknown error.
2050091     return (-1);
2050092 }
2050093 //
2050094 // Scan the inode table.
2050095 //
2050096 for (i = 0; i < INODE_MAX_SLOTS; i++)
2050097 {
2050098     if ((inode[i].sb ==
2050099         inode_mount_point->sb_attached)
2050100         && (inode[i].references > 0))
2050101     {
2050102         //
```

```
2050103         // At least one file is open inside the
2050104         // super block to
2050105         // release: cannot unmount.
2050106         //
2050107         inode_put (inode_mount_point);
2050108         errset (EBUSY);          // Device or resource
2050109         // busy.
2050110         return (-1);
2050111     }
2050112 }
2050113 //
2050114 // Can unmount: save and remove the super block
2050115 // memory;
2050116 // clear the mount point reference and put inode.
2050117 //
2050118 inode_mount_point->sb_attached->changed = 1;
2050119 sb_save (inode_mount_point->sb_attached);
2050120 //
2050121 inode_mount_point->sb_attached->device = 0;
2050122 inode_mount_point->sb_attached->inode_mounted_on = NULL;
2050123 inode_mount_point->sb_attached->blksize = 0;
2050124 inode_mount_point->sb_attached->options = 0;
2050125 //
2050126 inode_mount_point->sb_attached = NULL;
2050127 inode_mount_point->references = 0;
2050128 inode_put (inode_mount_point);
2050129 //
2050130 inode_put (inode_mount_point);
2050131 //
2050132 return (0);
2050133 }
```

94.8.48 kernel/lib_s/s_unlink.c



Si veda la sezione [87.62](#).

```
2060001 #include <kernel/fs.h>
2060002 #include <errno.h>
2060003 #include <kernel/proc.h>
2060004 #include <libgen.h>
2060005 #include <kernel/lib_s.h>
2060006 #include <kernel/lib_k.h>
2060007 //-----
2060008 int
2060009 s_unlink (pid_t pid, const char *path)
2060010 {
2060011     proc_t *ps;
2060012     inode_t *inode_unlink;
2060013     inode_t *inode_directory;
2060014     char path_unlink[PATH_MAX];
2060015     char path_copy[PATH_MAX];
2060016     char *path_directory;
2060017     char *name_unlink;
2060018     dev_t device;
2060019     off_t start;
2060020     char buffer[SB_MAX_ZONE_SIZE];
2060021     directory_t *dir = (directory_t *) buffer;
2060022     int status;
2060023     ssize_t size_read;
2060024     ssize_t size_written;
2060025     int d;          // Directory buffer index.
2060026     //
2060027     // Get process.
2060028     //
2060029     ps = proc_reference (pid);
2060030     //
2060031     // Get full paths.
2060032     //
2060033     path_full (path, ps->path_cwd, path_unlink);
2060034     strncpy (path_copy, path_unlink, PATH_MAX);
```

```
2060035 path_directory = dirname (path_copy);
2060036 //
2060037 // Get the inode to be unlinked.
2060038 //
2060039 inode_unlink = path_inode (pid, path_unlink);
2060040 if (inode_unlink == NULL)
2060041     {
2060042         return (-1);
2060043     }
2060044 //
2060045 // If it is a directory, verify that it is empty.
2060046 //
2060047 if (S_ISDIR (inode_unlink->mode))
2060048     {
2060049         if (!inode_dir_empty (inode_unlink))
2060050             {
2060051                 inode_put (inode_unlink);
2060052                 errset (ENOTEMPTY); // Directory not
2060053                 // empty.
2060054                 return (-1);
2060055             }
2060056     }
2060057 //
2060058 // Get the inode of the directory containing it.
2060059 //
2060060 inode_directory = path_inode (pid, path_directory);
2060061 if (inode_directory == NULL)
2060062     {
2060063         inode_put (inode_unlink);
2060064         return (-1);
2060065     }
2060066 //
2060067 // Check if something is mounted on the directory.
2060068 //
2060069 if (inode_directory->sb_attached != NULL)
2060070     {
2060071         //
```

```
2060072      // Must select the right directory.
2060073      //
2060074      device = inode_directory->sb_attached->device;
2060075      inode_put (inode_directory);
2060076      inode_directory = inode_get (device, 1);
2060077      if (inode_directory == NULL)
2060078          {
2060079              inode_put (inode_unlink);
2060080              return (-1);
2060081          }
2060082      }
2060083      //
2060084      // Check if write is allowed for the file system.
2060085      //
2060086      if (inode_directory->sb->options & MOUNT_RO)
2060087          {
2060088              errset (EROFS);    // Read-only file system.
2060089              return (-1);
2060090          }
2060091      //
2060092      // Verify access permissions for the directory. The
2060093      // number "3" means
2060094      // that the user must have access permission and
2060095      // write permission:
2060096      // "-wx" == 2+1 == 3.
2060097      //
2060098      status = inode_check (inode_directory, S_IFDIR, 3,
2060099                          ps->euid, ps->egid);
2060100      if (status != 0)
2060101          {
2060102              errset (EPERM);    // Operation not permitted.
2060103              inode_put (inode_unlink);
2060104              inode_put (inode_directory);
2060105              return (-1);
2060106          }
2060107      //
2060108      // Get the base name to be unlinked: this will alter
```

```
2060109 // the
2060110 // original path.
2060111 //
2060112 name_unlink = basename (path_unlink);
2060113 //
2060114 // Read the directory content and try to locate the
2060115 // item to unlink.
2060116 //
2060117 for (start = 0;
2060118      start < inode_directory->size;
2060119      start += inode_directory->sb->blksize)
2060120 {
2060121     size_read =
2060122         inode_file_read (inode_directory, start,
2060123                          buffer,
2060124                          inode_directory->sb->blksize,
2060125                          NULL);
2060126     if (size_read < sizeof (directory_t))
2060127     {
2060128         break;
2060129     }
2060130 //
2060131 // Scan the directory portion just read, for the
2060132 // item to unlink.
2060133 //
2060134 dir = (directory_t *) buffer;
2060135 //
2060136 for (d = 0; d < size_read;
2060137      d += (sizeof (directory_t)), dir++)
2060138 {
2060139     if ((dir->ino != 0)
2060140         &&
2060141         (strncmp
2060142          (dir->name, name_unlink, NAME_MAX) == 0))
2060143     {
2060144         //
2060145         // Found the corresponding item: unlink
```



```
2060146 // the inode.
2060147 //
2060148 dir->ino = 0;
2060149 //
2060150 // Update the directory inside the file
2060151 // system.
2060152 //
2060153 size_written =
2060154     inode_file_write (inode_directory,
2060155                       start, buffer, size_read);
2060156 if (size_written != size_read)
2060157     {
2060158         //
2060159         // Write problem: just tell.
2060160         //
2060161         k_printf
2060162             ("kernel alert: directory "
2060163              "write error!\n");
2060164     }
2060165 //
2060166 // Update directory inode and put inode.
2060167 // If the unlinked
2060168 // inode was a directory, the parent
2060169 // directory inode
2060170 // must reduce the file system link
2060171 // count.
2060172 //
2060173 if (S_ISDIR (inode_unlink->mode))
2060174     {
2060175         inode_directory->links--;
2060176     }
2060177 inode_directory->time = s_time (pid, NULL);
2060178 inode_directory->changed = 1;
2060179 inode_put (inode_directory);
2060180 //
2060181 // Reduce link inside unlinked inode and
2060182 // put inode.
```

```

2060183         //
2060184         inode_unlink->links--;
2060185         inode_unlink->changed = 1;
2060186         inode_unlink->time = s_time (pid, NULL);
2060187         inode_put (inode_unlink);
2060188         //
2060189         // Just return, as the work is done.
2060190         //
2060191         return (0);
2060192     }
2060193 }
2060194 }
2060195 //
2060196 // At this point, it was not possible to unlink the
2060197 // file.
2060198 //
2060199 inode_put (inode_unlink);
2060200 inode_put (inode_directory);
2060201 errset (EUNKNOWN);    // Unknown error.
2060202 return (-1);
2060203 }

```

94.8.49 kernel/lib_s/s_wait.c

<<

Si veda la sezione [87.63](#).

```

2070001 #include <kernel/proc.h>
2070002 #include <kernel/lib_s.h>
2070003 #include <errno.h>
2070004 //-----
2070005 pid_t
2070006 s_wait (pid_t pid, int *status)
2070007 {
2070008     pid_t parent = pid;
2070009     pid_t child;
2070010     int child_available = 0;
2070011     //

```

```
2070012 // Find a dead child process.
2070013 //
2070014 for (child = 1; child < PROCESS_MAX; child++)
2070015 {
2070016     if (proc_table[child].ppid == parent)
2070017     {
2070018         child_available = 1; // Child found!
2070019         if (proc_table[child].status == PROC_ZOMBIE)
2070020         {
2070021             break; // It is dead!
2070022         }
2070023     }
2070024 }
2070025 //
2070026 // If the index 'child' is a valid process number,
2070027 // a dead child was found.
2070028 //
2070029 if (child < PROCESS_MAX)
2070030 {
2070031     *status = proc_table[child].ret;
2070032     proc_available (child);
2070033     return (child);
2070034 }
2070035 else
2070036 {
2070037     if (child_available)
2070038     {
2070039         //
2070040         // There are child, but all alive.
2070041         //
2070042         // Go to sleep.
2070043         //
2070044         proc_table[parent].status = PROC_SLEEPING;
2070045         proc_table[parent].wakeup_events =
2070046             WAKEUP_EVENT_SIGNAL;
2070047         proc_table[parent].wakeup_signal = SIGCHLD;
2070048         return ((pid_t) 0);
```

```
2070049     }
2070050     else
2070051     {
2070052         //
2070053         // There are no child at all.
2070054         //
2070055         errset (ECHILD);
2070056         return ((pid_t) - 1);
2070057     }
2070058 }
2070059 }
```

94.8.50 kernel/lib_s/s_write.c

«

Si veda la sezione [87.64](#).

```
2080001 #include <kernel/proc.h>
2080002 #include <kernel/lib_s.h>
2080003 #include <errno.h>
2080004 #include <fcntl.h>
2080005 //-----
2080006 #define DEBUG 0
2080007 //-----
2080008 ssize_t
2080009 s_write (pid_t pid, int fdn, const void *buffer,
2080010         size_t count)
2080011 {
2080012     proc_t *ps;
2080013     fd_t *fd;
2080014     ssize_t size_written;
2080015     int status;
2080016     //
2080017     // Get process.
2080018     //
2080019     ps = proc_reference (pid);
2080020     //
2080021     // Get file descriptor.
```

```
2080022 //
2080023 fd = fd_reference (pid, &fdn);
2080024 if (fd == NULL || fd->file == NULL
2080025     || (fd->file->inode == NULL
2080026         && fd->file->sock == NULL))
2080027 {
2080028     //
2080029     // The file descriptor pointer is not valid.
2080030     //
2080031     errset (EBADF); // Bad file descriptor.
2080032     return ((ssize_t) - 1);
2080033 }
2080034 //
2080035 // Check if it is opened for write.
2080036 //
2080037 if (!(fd->file->oflags & O_WRONLY))
2080038 {
2080039     //
2080040     // The file is not opened for write.
2080041     //
2080042     errset (EINVAL); // Invalid argument.
2080043     return ((ssize_t) - 1);
2080044 }
2080045 //
2080046 // Check if it is a directory inode: a directory can
2080047 // be
2080048 // read as a file descriptor, but cannot be written.
2080049 //
2080050 if (fd->file->inode != NULL
2080051     && (fd->file->inode->mode & S_IFDIR))
2080052 {
2080053     errset (EISDIR); // Is a directory.
2080054     return ((ssize_t) - 1);
2080055 }
2080056 //
2080057 // It should be a valid type of file or socket to be
2080058 // written.
```

```
2080059 // Check if it is a file opened in append mode: if
2080060 // so, must move
2080061 // the write offset to the end.
2080062 //
2080063 if (fd->file->inode != NULL && (fd->fl_flags & O_APPEND))
2080064 {
2080065     fd->file->offset = fd->file->inode->size;
2080066 }
2080067 //
2080068 // Check the kind of socket/file to be written and
2080069 // write it.
2080070 //
2080071 if (fd->file->sock != NULL)
2080072 {
2080073     //
2080074     // Send it.
2080075     //
2080076     size_written = s_send (pid, fdn, buffer, count, 0);
2080077 }
2080078 else if (fd->file->inode->mode & S_IFBLK ||
2080079         fd->file->inode->mode & S_IFCHR)
2080080 {
2080081     //
2080082     // A device is to be written.
2080083     //
2080084     size_written =
2080085         dev_io (pid,
2080086                (dev_t) fd->file->inode->direct[0],
2080087                DEV_WRITE, (off_t) fd->file->offset,
2080088                (void *) buffer, count, NULL);
2080089 }
2080090 else if (fd->file->inode->mode & S_IFREG)
2080091 {
2080092     //
2080093     // A regular file is to be written.
2080094     //
2080095     size_written = inode_file_write (fd->file->inode,
```

```
2080096                                     fd->file->offset,
2080097                                     buffer, count);
2080098     }
2080099     else if (fd->file->inode->mode & S_IFIFO)
2080100     {
2080101         //
2080102         // A pipe is to be written.
2080103         //
2080104         size_written =
2080105             inode_pipe_write (fd->file->inode, buffer, count);
2080106         if (size_written == 0)
2080107         {
2080108             if (fd->file->inode->pipe_ref_read == 0)
2080109             {
2080110                 //
2080111                 // No read will be done anymore. Tell to
2080112                 // the process.
2080113                 //
2080114                 status = s_kill ((pid_t) 0, pid, SIGPIPE);
2080115                 if (status < 0)
2080116                 {
2080117                     errset (EPIPE);
2080118                     return ((ssize_t) - 1);
2080119                 }
2080120             else
2080121             {
2080122                 //
2080123                 // Wake up processes waiting to
2080124                 // read.
2080125                 //
2080126                 proc_wakeup_pipe_read (fd->file->inode);
2080127             }
2080128         }
2080129     else
2080130     {
2080131         //
2080132         // At the moment, nothing can be
```

```
2080133         // written.
2080134         //
2080135         if (fd->fl_flags & O_NONBLOCK)
2080136         {
2080137             //
2080138             // Cannot block.
2080139             //
2080140             errset (EAGAIN);
2080141             return ((ssize_t) - 1);
2080142         }
2080143     else
2080144     {
2080145         //
2080146         // Go to sleep.
2080147         //
2080148         proc_table[pid].status = PROC_SLEEPING;
2080149         proc_table[pid].ret = 0;
2080150         proc_table[pid].wakeup_inode =
2080151             fd->file->inode;
2080152         proc_table[pid].wakeup_events =
2080153             WAKEUP_EVENT_PIPE_WRITE;
2080154         if (DEBUG)
2080155             {
2080156                 k_printf
2080157                     ("[%s] PID %i goes to sleep "
2080158                      "waiting to write inside "
2080159                      "a pipe.\n", __FILE__, pid);
2080160             }
2080161     }
2080162 }
2080163 }
2080164 else
2080165 {
2080166     //
2080167     // Wake up processes waiting to read.
2080168     //
2080169     proc_wakeup_pipe_read (fd->file->inode);
```



```
2080170     }
2080171   }
2080172   else
2080173   {
2080174     //
2080175     // Unsupported file type.
2080176     //
2080177     errset (E_FILE_TYPE_UNSUPPORTED); // File type
2080178     // unsupported.
2080179     return ((ssize_t) - 1);
2080180   }
2080181   //
2080182   // Update the file descriptor internal offset, but
2080183   // only if it
2080184   // is a file.
2080185   //
2080186   if (fd->file->sock == NULL && size_written > 0)
2080187   {
2080188     fd->file->offset += size_written;
2080189   }
2080190   //
2080191   // Just return the size written, even if it is an
2080192   // error.
2080193   //
2080194   return (size_written);
2080195 }
```

94.9 os32: «kernel/main.h»

Si veda la sezione [93.13](#).

```
2090001 #ifndef _KERNEL_MAIN_H
2090002 #define _KERNEL_MAIN_H      1
2090003 //-----
2090004 #include <kernel/multiboot.h>
2090005 #include <stdint.h>
2090006 #include <sys/types.h>
```

```

2090007 //-----
2090008 void kmain (uint32_t magic, multiboot_t * mboot_data);
2090009 void menu (void);
2090010 pid_t run (char *path, char *argv[], char *envp[]);
2090011 //-----
2090012 #endif

```

94.9.1	kernel/main/build.h	1506
94.9.2	kernel/main/crt0.s	1506
94.9.3	kernel/main/kmain.c	1508
94.9.4	kernel/main/menu.c	1521
94.9.5	kernel/main/run.c	1522
94.9.6	kernel/main/stack.s	1523

94.9.1 kernel/main/build.h



Si veda la sezione [93.13](#).

```

2100001 #define BUILD_DATE "201108311936"

```

94.9.2 kernel/main/crt0.s



Si veda la sezione [84.2.2](#).

```

2110001 .extern kmain
2110002 .extern _k_stack_top
2110003 .extern _k_stack_bottom
2110004 .global kstartup
2110005 #####
2110006 #
2110007 # The kernel must be compiled as ELF, so that the
2110008 # bootloader (GRUB or SYSLINUX) can recognize it.
2110009 #

```

```
2110010 #-----
2110011 .section .text
2110012 #-----
2110013 #
2110014 # At the beginning there is the code, but inside the
2110015 # code there is also the multiboot header.
2110016 # This is why we start with a jump.
2110017 #
2110018 kstartup:
2110019     jmp start
2110020 #
2110021 # Here is the multiboot header, that must be placed
2110022 # near the beginning of the image-file, but aligned at
2110023 # a multiple of four bytes.
2110024 #
2110025 .align 4
2110026 multiboot_header:
2110027     .int 0x1BADB002           # magic
2110028     .int 0x00000007         # flags
2110029     .int -(0x1BADB002 + 0x00000007) # checksum
2110030 #
2110031 # Here starts really the code.
2110032 #
2110033 start:
2110034     #
2110035     # Set ESP at the stack bottom.
2110036     #
2110037     movl $_k_stack_bottom, %esp
2110038     #
2110039     # Reset flags inside EFLAGS, but must use the stack
2110040     # to make it.
2110041     #
2110042     pushl $0
2110043     popf
2110044     #
2110045     # Call function 'kmain()'. It is not the usual
2110046     # 'main()' because we need to pass some data, but
```

```

2110047     # it is not compatible with the
2110048     # standard 'main()' prototype.
2110049     #
2110050     # void kmain (uint32_t magic, multiboot_t *info);
2110051     #
2110052     pushl %ebx      # Pointer to the structure containing
2110053                   # data from the boot system.
2110054     pushl %eax      # Boot system signature.
2110055     #
2110056     call kmain      # Do the call.
2110057     #
2110058     # Halt procedure.
2110059     #
2110060     halt:
2110061         hlt         # If the function 'kmain()' return, this
2110062                   # instruction will halt the CPU.
2110063         jmp halt    # If the CPU will resume working, the
2110064                   # HLT instruction will be repeated.
2110065     #-----
2110066     .align 4
2110067     .section .data
2110068     #-----
2110069     .align 4
2110070     .section .bss
2110071     #-----

```

94.9.3 kernel/main/kmain.c



Si veda la sezione [93.13](#).

```

2120001     #include <kernel/main.h>
2120002     #include <kernel/main/build.h>
2120003     #include <kernel/lib_k.h>
2120004     #include <kernel/driver/tty.h>
2120005     #include <kernel/memory.h>
2120006     #include <stdlib.h>
2120007     #include <stdint.h>

```

```
2120008 #include <kernel/driver/screen.h>
2120009 #include <kernel/proc.h>
2120010 #include <kernel/lib_s.h>
2120011 #include <kernel/fs.h>
2120012 #include <unistd.h>
2120013 #include <stdint.h>
2120014 #include <kernel/driver/kbd.h>
2120015 #include <kernel/driver/ata.h>
2120016 #include <kernel/dev.h>
2120017 #include <kernel/blk.h>
2120018 #include <fcntl.h>
2120019 #include <string.h>
2120020 #include <limits.h>
2120021 #include <kernel/driver/pci.h>
2120022 #include <kernel/net/icmp.h>
2120023 #include <kernel/net/arp.h>
2120024 #include <kernel/net/route.h>
2120025 #include <kernel/net/tcp.h>
2120026 #include <kernel/driver/nic/ne2k.h>
2120027 #include <errno.h>
2120028 //-----
2120029 static char command[MAX_CANON];
2120030 //-----
2120031 void
2120032 kmain (uint32_t magic, multiboot_t * mboot_data)
2120033 {
2120034     int exit;
2120035     pid_t pid;
2120036     int counter;
2120037     char *exec_argv[2];
2120038     int status;
2120039     ssize_t count;
2120040     //
2120041     // 0xAC15FEFE == 172.21.254.254
2120042     //
2120043     h_addr_t ip_to_be_found = 0xAC15FEFE;
2120044     //
```

```
2120045 // Reset video and select the initial console.
2120046 //
2120047 tty_init ();
2120048 //
2120049 // Verify 'multiboot' data.
2120050 //
2120051 if (magic == 0x2BADB002)
2120052 {
2120053 //
2120054 // Save multiboot data.
2120055 //
2120056 mboot_save (mboot_data);
2120057 //
2120058 // Show compilation date and time, plus upper
2120059 // memory limit.
2120060 //
2120061 k_printf ("os32 build %s ram %i Kibyte\n",
2120062          BUILD_DATE, (int) multiboot.mem_upper);
2120063 //
2120064 // Show also the command line.
2120065 //
2120066 k_printf ("%s\n", multiboot.cmdline);
2120067 //
2120068 // Set 'mb_max', for memory allocation.
2120069 //
2120070 mb_size (multiboot.mem_upper * 1024);
2120071 //
2120072 // keyboard initialization.
2120073 //
2120074 kbd_load ();
2120075 //
2120076 // Block cache initialization.
2120077 //
2120078 blk_cache_init ();
2120079 //
2120080 // PCI bus initialization.
2120081 //
```

```
2120082     pci_init ();
2120083     //
2120084     // File system management initialization.
2120085     //
2120086     fs_init ();
2120087     //
2120088     // Set up processes.
2120089     //
2120090     proc_init ();
2120091     //
2120092     // Set up network.
2120093     //
2120094     net_init ();
2120095 }
2120096 else
2120097 {
2120098     //
2120099     // If it is not a multiboot loader, it is an
2120100     // error.
2120101     //
2120102     k_printf
2120103         ("os32 build %s. ERROR. no "
2120104         "\"multiboot\" header!\n", BUILD_DATE);
2120105     k_printf ("%s] system halted\n", __func__);
2120106     k_exit ();
2120107 }
2120108 //
2120109 // The kernel will run interactively.
2120110 //
2120111 k_printf
2120112     ("-----.\n"
2120113     "| `h' followed by [Enter] to show a menu |"
2120114     "\n"
2120115     "`-----'
2120116     "\n");
2120117 // menu ();
2120118
```

```
2120119 //
2120120 //
2120121 //
2120122 for (exit = 0; exit == 0;)
2120123 {
2120124 //
2120125 // While in kernel code, timer interrupt don't
2120126 // start the
2120127 // scheduler. The kernel must leave control to
2120128 // the scheduler
2120129 // via a null system call.
2120130 //
2120131 sys (SYS_0, NULL, 0);
2120132 //
2120133 // Back to work: read the keyboard from the TTY
2120134 // device.
2120135 //
2120136 count =
2120137     dev_io ((pid_t) 0, (dev_t) DEV_TTY, DEV_READ,
2120138             (off_t) 0, command, (size_t) MAX_CANON,
2120139             NULL);
2120140 //
2120141 // Check if there is a command: the kernel does
2120142 // not
2120143 // go to sleep.
2120144 //
2120145 if (count < 0)
2120146 {
2120147     if (errno == EAGAIN)
2120148     {
2120149 //
2120150 // No command is ready in the buffer
2120151 // keyboard.
2120152 //
2120153     continue;
2120154     }
2120155     else
```



```
2120156         {
2120157             k_perror (NULL);
2120158             continue;
2120159         }
2120160     }
2120161     if (count == 0)
2120162     {
2120163         //
2120164         // No command is ready in the buffer
2120165         // keyboard.
2120166         //
2120167         continue;
2120168     }
2120169     if (count == 1)
2120170     {
2120171         //
2120172         // Just [Enter] was pressed.
2120173         //
2120174         continue;
2120175     }
2120176     if (count > MAX_CANON)
2120177     {
2120178         //
2120179         // Something impossible!
2120180         //
2120181         continue;
2120182     }
2120183     //
2120184     // The last character is the "line delimiter"
2120185     // (new line et al.),
2120186     // or zero in case of EOF. The last character is
2120187     // replaced with
2120188     // zero, so that the command becomes a
2120189     // terminated string.
2120190     //
2120191     command[count - 1] = 0;
2120192     //
```

```
2120193 // A command was typed: start to check what it
2120194 // was.
2120195 //
2120196 if (strncmp (command, "1", MAX_CANON) == 0)
2120197 {
2120198     //
2120199     // Kill init.
2120200     //
2120201     s_kill ((pid_t) 0, (pid_t) 1, SIGKILL);
2120202 }
2120203 else if (strncmp (command, "2", MAX_CANON) == 0)
2120204 {
2120205     //
2120206     // Kill proc. 2
2120207     //
2120208     s_kill ((pid_t) 0, (pid_t) 2, SIGTERM);
2120209 }
2120210 else if (strncmp (command, "3", MAX_CANON) == 0)
2120211 {
2120212     //
2120213     // Kill proc. 3
2120214     //
2120215     s_kill ((pid_t) 0, (pid_t) 3, SIGTERM);
2120216 }
2120217 else if (strncmp (command, "4", MAX_CANON) == 0)
2120218 {
2120219     //
2120220     // Kill proc. 4
2120221     //
2120222     s_kill ((pid_t) 0, (pid_t) 4, SIGTERM);
2120223 }
2120224 else if (strncmp (command, "5", MAX_CANON) == 0)
2120225 {
2120226     //
2120227     // Kill proc. 5
2120228     //
2120229     s_kill ((pid_t) 0, (pid_t) 5, SIGTERM);
```

```
2120230     }
2120231     else if (strncmp (command, "6", MAX_CANON) == 0)
2120232     {
2120233         //
2120234         // Kill proc. 6
2120235         //
2120236         s_kill ((pid_t) 0, (pid_t) 6, SIGTERM);
2120237     }
2120238     else if (strncmp (command, "7", MAX_CANON) == 0)
2120239     {
2120240         //
2120241         // Kill proc. 7
2120242         //
2120243         s_kill ((pid_t) 0, (pid_t) 7, SIGTERM);
2120244     }
2120245     else if (strncmp (command, "8", MAX_CANON) == 0)
2120246     {
2120247         //
2120248         // Kill proc. 8
2120249         //
2120250         s_kill ((pid_t) 0, (pid_t) 8, SIGTERM);
2120251     }
2120252     else if (strncmp (command, "9", MAX_CANON) == 0)
2120253     {
2120254         //
2120255         // Kill proc. 9
2120256         //
2120257         s_kill ((pid_t) 0, (pid_t) 9, SIGTERM);
2120258     }
2120259     else if (strncmp (command, "A", MAX_CANON) == 0)
2120260     {
2120261         //
2120262         // Kill proc. 10
2120263         //
2120264         s_kill ((pid_t) 0, (pid_t) 10, SIGTERM);
2120265     }
2120266     else if (strncmp (command, "B", MAX_CANON) == 0)
```

```
2120267     {
2120268         //
2120269         // Kill proc. 11
2120270         //
2120271         s_kill ((pid_t) 0, (pid_t) 11, SIGTERM);
2120272     }
2120273 else if (strncmp (command, "C", MAX_CANON) == 0)
2120274     {
2120275         //
2120276         // Kill proc. 12
2120277         //
2120278         s_kill ((pid_t) 0, (pid_t) 12, SIGTERM);
2120279     }
2120280 else if (strncmp (command, "D", MAX_CANON) == 0)
2120281     {
2120282         //
2120283         // Kill proc. 13
2120284         //
2120285         s_kill ((pid_t) 0, (pid_t) 13, SIGTERM);
2120286     }
2120287 else if (strncmp (command, "E", MAX_CANON) == 0)
2120288     {
2120289         //
2120290         // Kill proc. 14
2120291         //
2120292         s_kill ((pid_t) 0, (pid_t) 14, SIGTERM);
2120293     }
2120294 else if (strncmp (command, "F", MAX_CANON) == 0)
2120295     {
2120296         //
2120297         // Kill proc. 15
2120298         //
2120299         s_kill ((pid_t) 0, (pid_t) 15, SIGTERM);
2120300     }
2120301 else if (strncmp (command, "a", MAX_CANON) == 0)
2120302     {
2120303         run ("/bin/aaa", NULL, NULL);
```

```
2120304     }
2120305     else if (strncmp (command, "b", MAX_CANON) == 0)
2120306     {
2120307         run ("/bin/bbb", NULL, NULL);
2120308     }
2120309     else if (strncmp (command, "c", MAX_CANON) == 0)
2120310     {
2120311         run ("/bin/ccc", NULL, NULL);
2120312     }
2120313     else if (strncmp (command, "f", MAX_CANON) == 0)
2120314     {
2120315         pid = fork ();
2120316         if (pid == -1)
2120317         {
2120318             k_perror (NULL);
2120319         }
2120320         else if (pid == 0)
2120321         {
2120322             //
2120323             // Get child real pid.
2120324             //
2120325             pid = getpid ();
2120326             //
2120327             // Please note that the child is no more
2120328             // a kernel, and can access to process
2120329             // system calls.
2120330             //
2120331             for (counter = 0; counter < 60; counter++)
2120332             {
2120333                 z_printf ("%1x", (int) pid);
2120334                 sleep (1);
2120335             }
2120336         }
2120337         else
2120338         {
2120339             z_printf ("io sono il genitore di %i\n", pid);
2120340         }
```

```
2120341     }
2120342     else if (strncmp (command, "g", MAX_CANON) == 0)
2120343     {
2120344         gdt_print (&gdt_register, 0, 20);
2120345     }
2120346     else if (strncmp (command, "G", MAX_CANON) == 0)
2120347     {
2120348         gdt_print (&gdt_register, 21, 41);
2120349     }
2120350     else if (strncmp (command, "h", MAX_CANON) == 0)
2120351     {
2120352         menu ();
2120353     }
2120354     else if (strncmp (command, "i", MAX_CANON) == 0)
2120355     {
2120356         idt_print (&idt_register, 0, 20);
2120357     }
2120358     else if (strncmp (command, "I", MAX_CANON) == 0)
2120359     {
2120360         idt_print (&idt_register, 21, 41);
2120361     }
2120362     else if (strncmp (command, "m", MAX_CANON) == 0)
2120363     {
2120364         mb_print ();
2120365     }
2120366     else if (strncmp (command, "n", MAX_CANON) == 0)
2120367     {
2120368         inode_print ();
2120369     }
2120370     else if (strncmp (command, "p", MAX_CANON) == 0)
2120371     {
2120372         proc_print ();
2120373     }
2120374     else if (strncmp (command, "s", MAX_CANON) == 0)
2120375     {
2120376         sb_print ();
2120377     }
```

```
2120378     else if (strncmp (command, "t", MAX_CANON) == 0)
2120379     {
2120380         k_printf ("clock: %lli time: %i\n",
2120381                 (long long int) k_clock (),
2120382                 (int) k_time (NULL));
2120383     }
2120384     else if (strncmp (command, "T", MAX_CANON) == 0)
2120385     {
2120386         while (1)
2120387         {
2120388             k_printf ("clock: %lli time: %i\n",
2120389                     (long long int) k_clock (),
2120390                     (int) k_time (NULL));
2120391         }
2120392     }
2120393     else if (strncmp (command, "w", MAX_CANON) == 0)
2120394     {
2120395         status = s_open ((pid_t) 0, "/tmp/test",
2120396                         O_WRONLY | O_CREAT |
2120397                         O_TRUNC, 0644);
2120398         //
2120399         if (status >= 0)
2120400         {
2120401             status = s_close ((pid_t) 0, status);
2120402             if (status != 0)
2120403             {
2120404                 k_perror (NULL);
2120405             }
2120406         }
2120407     }
2120408     else if (strncmp (command, "x", MAX_CANON) == 0)
2120409     {
2120410         //
2120411         // Load init.
2120412         //
2120413         exec_argv[0] = "/bin/init";
2120414         exec_argv[1] = NULL;
```

```
2120415     pid = run ("/bin/init", exec_argv, NULL);
2120416     //
2120417     // Just sleep.
2120418     //
2120419     while (1)
2120420     {
2120421         sys (SYS_0, NULL, 0);
2120422     }
2120423 }
2120424 else if (strncmp (command, "q", MAX_CANON) == 0)
2120425 {
2120426     k_printf ("System halted!\n");
2120427     return;
2120428 }
2120429 else if (strncmp (command, "y", MAX_CANON) == 0)
2120430 {
2120431     // icmp_test3 ();
2120432     // icmp_test2 ();
2120433     // ip_test2 ();
2120434     // ip_test ();
2120435     tcp_test ();
2120436 }
2120437 else if (strncmp (command, "r", MAX_CANON) == 0)
2120438 {
2120439     arp_print ();
2120440 }
2120441 else if (strncmp (command, "R", MAX_CANON) == 0)
2120442 {
2120443     arp_request (ip_to_be_found);
2120444 }
2120445 }
2120446 }
```


94.9.4 kernel/main/menu.c



Si veda la sezione [93.13](#).

```

2130001 #include <kernel/main.h>
2130002 #include <kernel/lib_k.h>
2130003 //-----
2130004 void
2130005 menu (void)
2130006 {
2130007     k_printf
2130008     ("-----."
2130009     "\n"
2130010     "| h      show this menu          .-----.|"
2130011     "\n"
2130012     "| t      show internal timer values |all      ||"
2130013     "\n"
2130014     "| f      fork the kernel          |commands ||"
2130015     "\n"
2130016     "| m      memory map (HEX)         |followed  ||"
2130017     "\n"
2130018     "| g|G    show GDT table first 21+21 items|by [Enter] ||"
2130019     "\n"
2130020     "| i|I    show IDT table first 21+21 items`-----'|"
2130021     "\n"
2130022     "| p      process status list          |"
2130023     "\n"
2130024     "| s      super block list            |"
2130025     "\n"
2130026     "| n      list of active inodes        |"
2130027     "\n"
2130028     "| 1..9   kill process  1 to 9         |"
2130029     "\n"
2130030     "| A..F   kill process 10 to 15        |"
2130031     "\n"
2130032     "| a..c   run programs  `/bin/aaa' to  `/bin/ccc'"
2130033     "\n"
2130034     "|           in paralel                |"

```

```

2130035     "\n"
2130036     "| r      ARP table                               |"
2130037     "\n"
2130038     "| x      exit kernel interaction, start '/bin/init' |"
2130039     "\n"
2130040     "| q      quit kernel                               |"
2130041     "\n"
2130042     "\ `-----' "
2130043     "\n");
2130044 }

```

94.9.5 kernel/main/run.c

«

Si veda la sezione [93.13](#).

```

2140001 #include <kernel/main.h>
2140002 #include <kernel/proc.h>
2140003 #include <kernel/lib_k.h>
2140004 #include <unistd.h>
2140005 //-----
2140006 pid_t
2140007 run (char *path, char *argv[], char *envp[])
2140008 {
2140009     pid_t pid;
2140010     //
2140011     pid = fork ();
2140012     if (pid == -1)
2140013     {
2140014         k_perror (NULL);
2140015     }
2140016     else if (pid == 0)
2140017     {
2140018         execve (path, argv, envp);
2140019         z_perror (NULL);
2140020         _exit (0);
2140021     }
2140022     return (pid);

```

2140023

}

94.9.6 kernel/main/stack.s



Si veda la sezione [93.13](#).

```

2150001  .global _k_stack_top
2150002  .global _k_stack_bottom
2150003  #####
2150004  #
2150005  # Kernel stack size. The value 0x010000 is equal to
2150006  # 1 Mibyte.
2150007  # Please note that if the kernel stack is too little,
2150008  # there is no way to check it.
2150009  #
2150010  .equ STACK_SIZE, 0x0100000
2150011  #-----
2150012  .align 4
2150013  .section .bss
2150014  #-----
2150015  #
2150016  # At the end is placed the space for the kernel stack,
2150017  # with no initialization.
2150018  #
2150019  _k_stack_top:
2150020  .space STACK_SIZE
2150021  _k_stack_bottom:
2150022  #-----

```

94.10 os32: «kernel/memory.h»



Si veda la sezione [93.14](#).

```

2160001  #ifndef _KERNEL_MEMORY_H
2160002  #define _KERNEL_MEMORY_H      1
2160003  //-----

```

```

2160004 #include <stdint.h>
2160005 #include <stddef.h>
2160006 #include <sys/types.h>
2160007 //-----
2160008 #define MEM_BLOCK_SIZE 0x1000 // 4 Ki/block
2160009 #define MEM_MAX_BLOCKS 0x100000 // 1 Mi blocks
2160010 // = 4 Gbyte
2160011
2160012 extern uint32_t mb_table[MEM_MAX_BLOCKS / 32]; // [1]
2160013 //
2160014 // [1] Memory blocks map.
2160015 //
2160016 extern unsigned int mb_max; // Memory blocks max.
2160017 //-----
2160018 typedef unsigned int addr_t;
2160019 //-----
2160020 uint32_t *mb_reference (void);
2160021 ssize_t mb_alloc (addr_t address, size_t size);
2160022 void mb_free (addr_t address, size_t size);
2160023 int mb_reduce (addr_t address, size_t new, size_t previous);
2160024 void mb_clean (addr_t address, size_t size);
2160025 addr_t mb_alloc_size (size_t size);
2160026 void mb_print (void);
2160027 void mb_size (size_t size);
2160028 //-----
2160029 #endif

```

94.10.1	kernel/memory/mb_alloc.c	1525
94.10.2	kernel/memory/mb_alloc_size.c	1528
94.10.3	kernel/memory/mb_clean.c	1531
94.10.4	kernel/memory/mb_free.c	1531
94.10.5	kernel/memory/mb_print.c	1534
94.10.6	kernel/memory/mb_public.c	1536

94.10.7	kernel/memory/mb_reduce.c	1536
94.10.8	kernel/memory/mb_reference.c	1538
94.10.9	kernel/memory/mb_size.c	1538

94.10.1 kernel/memory/mb_alloc.c

Si veda la sezione [93.14](#).



```
2170001 #include <kernel/memory.h>
2170002 #include <kernel/ibm_i386.h>
2170003 #include <sys/os32.h>
2170004 #include <kernel/lib_k.h>
2170005 //-----
2170006 #define DEBUG 0
2170007 //-----
2170008 static int mb_block_set1 (int block);
2170009 //-----
2170010 ssize_t
2170011 mb_alloc (addr_t address, size_t size)
2170012 {
2170013     unsigned int bstart;
2170014     unsigned int bsize;
2170015     unsigned int bend;
2170016     unsigned int i;
2170017     ssize_t allocated = 0;
2170018     addr_t block_address;
2170019     //
2170020     if (size == 0)
2170021     {
2170022         //
2170023         // Zero means nothing.
2170024         //
2170025         allocated = 0;
2170026         return (allocated);
2170027     }
2170028     //
```

```
2170029 // Show what was requested.
2170030 //
2170031 if (DEBUG)
2170032 {
2170033     k_printf ("%s(%i, %zi)", __func__,
2170034             (unsigned int) address, size);
2170035 }
2170036 //
2170037 // Calculate starting block of memory.
2170038 //
2170039 bstart = address / MEM_BLOCK_SIZE;
2170040 //
2170041 // Calculating size in term of blocks.
2170042 //
2170043 if (size % MEM_BLOCK_SIZE)
2170044 {
2170045     bsize = size / MEM_BLOCK_SIZE + 1;
2170046 }
2170047 else
2170048 {
2170049     bsize = size / MEM_BLOCK_SIZE;
2170050 }
2170051 //
2170052 // Calculate the block number after the
2170053 // end of the requested memory area.
2170054 //
2170055 bend = bstart + bsize;
2170056 //
2170057 // Scan the memory map and allocate.
2170058 //
2170059 for (i = bstart; i < bend; i++)
2170060 {
2170061     if (mb_block_set1 (i))
2170062     {
2170063         allocated += MEM_BLOCK_SIZE;
2170064     }
2170065     else
```

```
2170066     {
2170067         block_address = i;
2170068         block_address += MEM_BLOCK_SIZE;
2170069         k_printf
2170070             ("[%s] Kernel alert: mem block "
2170071              "%x, at address "
2170072              "%x, already allocated!\n", __FILE__, i,
2170073              (unsigned int) block_address);
2170074         break;
2170075     }
2170076 }
2170077 //
2170078 //
2170079 //
2170080 return (allocated);
2170081 }
2170082
2170083 //-----
2170084 static int
2170085 mb_block_set1 (int block)
2170086 {
2170087     int i = block / 32;
2170088     int j = block % 32;
2170089     uint32_t mask = 0x80000000 >> j;
2170090     if (mb_table[i] & mask)
2170091     {
2170092         return (0);        // The block is already set to
2170093         // 1 inside the map!
2170094     }
2170095     else
2170096     {
2170097         mb_table[i] = mb_table[i] | mask;
2170098         return (1);
2170099     }
2170100 }
```

94.10.2 kernel/memory/mb_alloc_size.c



Si veda la sezione [93.14](#).

```
2180001 #include <kernel/memory.h>
2180002 #include <kernel/ibm_i386.h>
2180003 #include <kernel/lib_k.h>
2180004 #include <sys/os32.h>
2180005 #include <errno.h>
2180006 //-----
2180007 #define DEBUG 0
2180008 //-----
2180009 static int mb_block_status (int block);
2180010 //-----
2180011 addr_t
2180012 mb_alloc_size (size_t size)
2180013 {
2180014     unsigned int bsize;
2180015     unsigned int i;
2180016     unsigned int j;
2180017     unsigned int found = 0;
2180018     addr_t alloc_addr;
2180019     ssize_t alloc_size;
2180020     //
2180021     //
2180022     //
2180023     if (size == 0)
2180024     {
2180025         errset (EINVAL);
2180026         return ((addr_t) 0);
2180027     }
2180028     //
2180029     // Show what was requested.
2180030     //
2180031     if (DEBUG)
2180032     {
2180033         k_printf ("%s(%zi)", __func__, size);
2180034     }
```



```
2180035 //
2180036 // Calculate block size.
2180037 //
2180038 if (size % MEM_BLOCK_SIZE)
2180039 {
2180040     bsize = size / MEM_BLOCK_SIZE + 1;
2180041 }
2180042 else
2180043 {
2180044     bsize = size / MEM_BLOCK_SIZE;
2180045 }
2180046 //
2180047 // Scan for a contiguous space in memory.
2180048 //
2180049 for (i = 0; i < (mb_max - bsize) && !found; i++)
2180050 {
2180051     for (j = 0; j < bsize; j++)
2180052     {
2180053         found = !mb_block_status (i + j);
2180054         if (!found)
2180055         {
2180056             i += j;
2180057             break;
2180058         }
2180059     }
2180060 }
2180061 //
2180062 // If the space was found, allocate it.
2180063 //
2180064 if (found && (j == bsize))
2180065 {
2180066     alloc_addr = i - 1;
2180067     alloc_addr *= MEM_BLOCK_SIZE;
2180068     alloc_size = bsize * MEM_BLOCK_SIZE;
2180069     alloc_size =
2180070     mb_alloc (alloc_addr, (size_t) alloc_size);
2180071 //
```

```
2180072     if (alloc_size <= 0)
2180073     {
2180074         errset (ENOMEM);
2180075         return ((addr_t) 0);
2180076     }
2180077     else if (alloc_size < size)
2180078     {
2180079         mb_free (alloc_addr, (size_t) alloc_size);
2180080         errset (ENOMEM);
2180081         return ((addr_t) 0);
2180082     }
2180083     else
2180084     {
2180085         //
2180086         // Clean memory before return.
2180087         //
2180088         mb_clean (alloc_addr, (size_t) alloc_size);
2180089         //
2180090         //
2180091         //
2180092         return (alloc_addr);
2180093     }
2180094 }
2180095 else
2180096 {
2180097     errset (ENOMEM);
2180098     return ((addr_t) 0);
2180099 }
2180100 }
2180101
2180102 //-----
2180103 static int
2180104 mb_block_status (int block)
2180105 {
2180106     int i = block / 32;
2180107     int j = block % 32;
2180108     uint32_t mask = 0x80000000 >> j;
```

```
2180109     return ((int) (mb_table[i] & mask));
2180110 }
```

94.10.3 kernel/memory/mb_clean.c

Si veda la sezione [93.14](#).

```
2190001 #include <kernel/memory.h>
2190002 //-----
2190003 void
2190004 mb_clean (addr_t address, size_t size)
2190005 {
2190006     unsigned int i;
2190007     char *mem;
2190008     //
2190009     mem = (char *) address;
2190010     //
2190011     for (i = 0; i < size; i++)
2190012     {
2190013         mem[i] = 0;
2190014     }
2190015 }
```

94.10.4 kernel/memory/mb_free.c

Si veda la sezione [93.14](#).

```
2200001 #include <kernel/memory.h>
2200002 #include <kernel/ibm_i386.h>
2200003 #include <sys/os32.h>
2200004 #include <kernel/lib_k.h>
2200005 //-----
2200006 #define DEBUG 0
2200007 //-----
2200008 static int mb_block_set0 (int block);
2200009 //-----
2200010 void
```

```
2200011 mb_free (addr_t address, size_t size)
2200012 {
2200013     unsigned int bstart;
2200014     unsigned int bsize;
2200015     unsigned int bend;
2200016     unsigned int i;
2200017     addr_t block_address;
2200018     //
2200019     // k_printf ("releasing 0x%x, size 0x%x\n", (int)
2200020     // address,
2200021     // (int) size);
2200022     //
2200023     if (size == 0)
2200024     {
2200025         //
2200026         // Zero means nothing.
2200027         //
2200028         return;
2200029     }
2200030     //
2200031     // Show what was requested.
2200032     //
2200033     if (DEBUG)
2200034     {
2200035         k_printf ("%s(%i, %zi)", __func__,
2200036                 (unsigned int) address, size);
2200037     }
2200038     //
2200039     // Calculate size in term of blocks.
2200040     //
2200041     if (size % MEM_BLOCK_SIZE)
2200042     {
2200043         bsize = size / MEM_BLOCK_SIZE + 1;
2200044     }
2200045     else
2200046     {
2200047         bsize = size / MEM_BLOCK_SIZE;
```

```
2200048     }
2200049     //
2200050     // Calculate start address in term of blocks.
2200051     //
2200052     bstart = address / MEM_BLOCK_SIZE;
2200053     //
2200054     // Calculate end address, in term of blocks.
2200055     // This address is after the memory area to
2200056     // be released.
2200057     //
2200058     bend = bstart + bsize;
2200059     //
2200060     // Scan the memory map to free memory blocks.
2200061     //
2200062     for (i = bstart; i < bend; i++)
2200063     {
2200064         if (mb_block_set0 (i))
2200065         {
2200066             ;
2200067         }
2200068         else
2200069         {
2200070             block_address = i;
2200071             block_address *= MEM_BLOCK_SIZE;
2200072             k_printf
2200073                 ("%s] Kernel alert: mem block "
2200074                 "0x%x, at address "
2200075                 "0x%x, already released!\n", __FILE__, i,
2200076                 (unsigned int) block_address);
2200077         }
2200078     }
2200079 }
2200080
2200081 //-----
2200082 static int
2200083 mb_block_set0 (int block)
2200084 {
```

```
2200085     int i = block / 32;
2200086     int j = block % 32;
2200087     uint32_t mask = 0x80000000 >> j;
2200088     if (mb_table[i] & mask)
2200089     {
2200090         mb_table[i] = mb_table[i] & ~mask;
2200091         return (1);
2200092     }
2200093     else
2200094     {
2200095         return (0);           // The block is already set to
2200096         // 0 inside the map!
2200097     }
2200098 }
```

94.10.5 kernel/memory/mb_print.c

<<

Si veda la sezione [93.14](#).

```
2210001 #include <kernel/memory.h>
2210002 #include <kernel/ibm_i386.h>
2210003 #include <sys/os32.h>
2210004 #include <kernel/lib_k.h>
2210005 #include <kernel/multiboot.h>
2210006 //-----
2210007 void
2210008 mb_print (void)
2210009 {
2210010     unsigned int block;
2210011     unsigned int blocks =
2210012         (multiboot.mem_upper * 1024 / MEM_BLOCK_SIZE);
2210013     int i;
2210014     int j;
2210015     uint32_t mask;
2210016     unsigned int start = 0;
2210017     unsigned int stop = 0;
2210018     unsigned int status = 0;
```

```
2210019 //
2210020 k_printf ("Hex mem map, blocks of %x:", MEM_BLOCK_SIZE);
2210021 //
2210022 for (block = 0; block < blocks; block++)
2210023 {
2210024     i = block / 32;
2210025     j = block % 32;
2210026     mask = 0x80000000 >> j;
2210027     if (mb_table[i] & mask)
2210028     {
2210029         //
2210030         // Allocated block
2210031         //
2210032         if (status == 0)
2210033         {
2210034             status = 1;
2210035             start = block;
2210036         }
2210037     }
2210038     else
2210039     {
2210040         //
2210041         // Not allocated block.
2210042         //
2210043         if (status == 1)
2210044         {
2210045             status = 0;
2210046             stop = block;
2210047         }
2210048     }
2210049     //
2210050     //
2210051     //
2210052     if (stop > 0)
2210053     {
2210054         k_printf (" %x-%x", start, stop);
2210055         start = 0;
```

```
2210056         stop = 0;
2210057     }
2210058 }
2210059 k_printf ("\n");
2210060 }
```

94.10.6 kernel/memory/mb_public.c

<<

Si veda la sezione [93.14](#).

```
2220001 #include <kernel/memory.h>
2220002 #include <stdint.h>
2220003 //-----
2220004 uint32_t mb_table[MEM_MAX_BLOCKS / 32]; // Memory
2220005                                           // blocks map.
2220006 unsigned int mb_max = MEM_MAX_BLOCKS; // Memory
2220007                                           // blocks max.
2220008 //-----
```

94.10.7 kernel/memory/mb_reduce.c

<<

Si veda la sezione [93.14](#).

```
2230001 #include <kernel/memory.h>
2230002 #include <kernel/ibm_i386.h>
2230003 #include <sys/os32.h>
2230004 #include <errno.h>
2230005 //-----
2230006 int
2230007 mb_reduce (addr_t address, size_t new, size_t previous)
2230008 {
2230009     addr_t start;
2230010     addr_t end;
2230011     size_t size;
2230012     //
2230013     //
2230014     //
```



```
2230015     if (new > previous)
2230016     {
2230017         //
2230018         // We are reducing, not extending!
2230019         //
2230020         errset (EINVAL);
2230021         return (-1);
2230022     }
2230023     //
2230024     //
2230025     //
2230026     if (new == previous)
2230027     {
2230028         //
2230029         // Nothing to do.
2230030         //
2230031         return (0);
2230032     }
2230033     //
2230034     // Correct sizes to conform to memory blocks.
2230035     //
2230036     start = address + new;
2230037     if (start % MEM_BLOCK_SIZE)
2230038     {
2230039         start /= MEM_BLOCK_SIZE;
2230040         start++;
2230041         start *= MEM_BLOCK_SIZE;
2230042     }
2230043     //
2230044     end = address + previous;
2230045     end /= MEM_BLOCK_SIZE;
2230046     end *= MEM_BLOCK_SIZE;
2230047     //
2230048     size = end - start;
2230049     //
2230050     // Finally release the extra memory, no more used.
2230051     //
```

```
2230052     mb_free (start, size);
2230053     //
2230054     // ok.
2230055     //
2230056     return (0);
2230057 }
```

94.10.8 kernel/memory/mb_reference.c

<<

Si veda la sezione [93.14](#).

```
2240001 #include <stdint.h>
2240002 #include <kernel/memory.h>
2240003 //-----
2240004 uint32_t *
2240005 mb_reference (void)
2240006 {
2240007     return mb_table;
2240008 }
```

94.10.9 kernel/memory/mb_size.c

<<

Si veda la sezione [93.14](#).

```
2250001 #include <kernel/memory.h>
2250002 //-----
2250003 void
2250004 mb_size (size_t size)
2250005 {
2250006     mb_max = size / MEM_BLOCK_SIZE;
2250007 }
```

94.11 os32: «kernel/multiboot.h»



Si veda la sezione [93.15](#).

```
2260001 #ifndef _KERNEL_MULTIBOOT_H
2260002 #define _KERNEL_MULTIBOOT_H      1
2260003 //-----
2260004 #include <inttypes.h>
2260005 //-----
2260006 #define MBOOT_CMDLINE_MAX      4096
2260007 #define MBOOT_CMDLINE_OPTION_MAX 1024
2260008 #define MBOOT_CMDLINE_ARGUMENTS_MAX 32
2260009 //-----
2260010 typedef struct
2260011 {
2260012     uint32_t flags;
2260013     uint32_t mem_lower;
2260014     uint32_t mem_upper;
2260015     uint32_t boot_device;
2260016     char *cmdline;
2260017 } multiboot_t;
2260018 //
2260019 typedef struct
2260020 {
2260021     uint32_t flags;
2260022     uint32_t mem_lower;
2260023     uint32_t mem_upper;
2260024     uint32_t boot_device;
2260025     char cmdline[MBOOT_CMDLINE_MAX];
2260026 } multiboot_save_t;
2260027 //
2260028 extern multiboot_save_t multiboot;
2260029 //-----
2260030 void mboot_save (multiboot_t * mboot_data);
2260031 char **mboot_cmdline_opt (const char *opt,
2260032                          const char *delim);
2260033 //-----
2260034 #endif
```

94.11.1	kernel/multiboot/mboot_cmdline_opt.c	1540
94.11.2	kernel/multiboot/mboot_public.c	1543
94.11.3	kernel/multiboot/mboot_save.c	1543

94.11.1 kernel/multiboot/mboot_cmdline_opt.c



Si veda la sezione [93.15](#).

```
2270001 #include <stddef.h>
2270002 #include <kernel/multiboot.h>
2270003 #include <kernel/lib_k.h>
2270004 #include <string.h>
2270005 #include <errno.h>
2270006 //-----
2270007 char **
2270008 mboot_cmdline_opt (const char *opt, const char *delim)
2270009 {
2270010     static char option[MBOOT_CMDLINE_OPTION_MAX];
2270011     static char *argument[MBOOT_CMDLINE_ARGUMENTS_MAX];
2270012     char *a;
2270013     char *z;
2270014     char *t;
2270015     int i;
2270016     size_t size;
2270017     //
2270018     // Check input.
2270019     //
2270020     if (opt == NULL)
2270021     {
2270022         errset (EINVAL);
2270023         return (NULL);
2270024     }
2270025     //
2270026     // Find the option.
2270027     //
2270028     a = strstr (multiboot.cmdline, opt);
```

```
2270029     if (a == NULL)
2270030     {
2270031         return (NULL);
2270032     }
2270033     //
2270034     // Find the end of the option: might be a space or
2270035     // the end of the
2270036     // string.
2270037     //
2270038     z = strpbrk (a, " \t\n");
2270039     //
2270040     // Copy the option inside the static array
2270041     // 'option[]'.
2270042     //
2270043     if (z == NULL)
2270044     {
2270045         strncpy (option, a, MBOOT_CMDLINE_OPTION_MAX - 1);
2270046         option[MBOOT_CMDLINE_OPTION_MAX - 1] = 0;
2270047     }
2270048     else
2270049     {
2270050         size = (uintptr_t) z - (uintptr_t) a;
2270051         strncpy (option, a, size);
2270052         option[size] = 0;
2270053     }
2270054     //
2270055     // Find the option name, to be saved as the first
2270056     // argument.
2270057     //
2270058     t = strtok (option, "=");
2270059     if (t == NULL)
2270060     {
2270061         errset (EUNKNOWN);
2270062         return (NULL);
2270063     }
2270064     argument[0] = t;
2270065     //
```

```
2270066 // If there is no delimiter, replace it with a
2270067 // string containing
2270068 // just a space.
2270069 //
2270070 if (delim == NULL)
2270071 {
2270072     delim = " ";
2270073 }
2270074 //
2270075 for (i = 1; i < MBOOT_CMDLINE_ARGUMENTS_MAX; i++)
2270076 {
2270077     t = strtok (NULL, delim);
2270078     if (t == NULL)
2270079     {
2270080         //
2270081         // The argument will be an empty string,
2270082         // taken from
2270083         // the end of the option string.
2270084         //
2270085         argument[i] =
2270086             &option[MBOOT_CMDLINE_OPTION_MAX - 1];
2270087     }
2270088     else
2270089     {
2270090         argument[i] = t;
2270091     }
2270092 }
2270093 //
2270094 // Return.
2270095 //
2270096 return (argument);
2270097 }
```

94.11.2 kernel/multiboot/mboot_public.c



Si veda la sezione [93.15](#).

```
2280001 #include <kernel/multiboot.h>
2280002 //-----
2280003 multiboot_save_t multiboot;
```

94.11.3 kernel/multiboot/mboot_save.c



Si veda la sezione [93.15](#).

```
2290001 #include <kernel/multiboot.h>
2290002 #include <string.h>
2290003 #include <kernel/lib_k.h>
2290004 //-----
2290005 void
2290006 mboot_save (multiboot_t * mboot_data)
2290007 {
2290008     multiboot.flags = mboot_data->flags;
2290009     //
2290010     if ((mboot_data->flags & 1) > 0)
2290011     {
2290012         multiboot.mem_lower = mboot_data->mem_lower;
2290013         multiboot.mem_upper = mboot_data->mem_upper;
2290014     }
2290015     if ((mboot_data->flags & 2) > 0)
2290016     {
2290017         multiboot.boot_device = mboot_data->boot_device;
2290018     }
2290019     if ((mboot_data->flags & 4) > 0)
2290020     {
2290021         strncpy (multiboot.cmdline, mboot_data->cmdline,
2290022                 MBOOT_CMDLINE_MAX);
2290023     }
2290024     else
2290025     {
2290026         memset (multiboot.cmdline, 0, MBOOT_CMDLINE_MAX);
```

```
2290027     }
2290028 }
```

94.12 os32: «kernel/net.h»

<<

Si veda la sezione [93.17](#).

```
2300001 #ifndef _KERNEL_NET_H
2300002 #define _KERNEL_NET_H      1
2300003 //-----
2300004 //
2300005 // WARNING: please remember to limit the max packets
2300006 // size to be transmitted, to a value that ensure no
2300007 // fragmentation at IPv4 level.
2300008 //
2300009 //-----
2300010 #include <stdint.h>
2300011 #include <sys/types.h>
2300012 #include <netinet/in.h>
2300013 #include <lib/sys/os32.h>
2300014 #include <netinet/ip.h>
2300015 #include <arpa/inet.h>
2300016 //-----
2300017 #define NET_PROT_IP        0x0800
2300018 #define NET_PROT_ARP      0x0806
2300019 #define NET_PROT_RARP     0x8035
2300020 //
2300021 #define NET_ETHERNET_ADDRESS_LENGTH  6
2300022 #define NET_IP_ADDRESS_LENGTH       4
2300023 //
2300024 #define NET_DEV_NULL                0x0000
2300025 #define NET_DEV_LOOP                 0x0100
2300026 #define NET_DEV_LOOPBACK             0x0101
2300027 #define NET_DEV_ETH                  0x0200
2300028 #define NET_DEV_ETH_NE2K             0x0201
2300029 //
2300030 #define NET_MAX_DEVICES              4
```



```
2300031 #define NET_MAX_BUFFERS 64
2300032 //
2300033 #define NET_ETHERNET_MIN_PACKET_SIZE 46
2300034 #define NET_ETHERNET_MAX_PACKET_SIZE 1500
2300035 #define NET_ETHERNET_HEADER_SIZE 14
2300036 #define NET_ETHERNET_MAX_FRAME_SIZE \
2300037     (NET_ETHERNET_MAX_PACKET_SIZE \
2300038     +NET_ETHERNET_HEADER_SIZE)
2300039 //
2300040 #define NET_IP_MIN_HEADER_SIZE 20
2300041 #define NET_IP_MIN_PACKET_SIZE 576
2300042 #define NET_IP_MAX_PACKET_SIZE \
2300043     NET_ETHERNET_MAX_PACKET_SIZE // [1]
2300044 #define NET_IP_MAX_DATA_SIZE \
2300045     (NET_IP_MAX_PACKET_SIZE \
2300046     -NET_IP_MIN_HEADER_SIZE)
2300047 //
2300048 // [1] The IP max packet size should be 65535, but as
2300049 // os32 does not accept fragmented packets, the
2300050 // maximum value depends on the Ethernet packet
2300051 // size. But this limitation might change, so
2300052 // there is also the value NET_MTU, that depends
2300053 // from the real physical network limitations.
2300054 //
2300055 #define NET_MTU NET_ETHERNET_MAX_PACKET_SIZE
2300056 //-----
2300057 //
2300058 // IP unfragmented packet, but it is not known if there
2300059 // are IP options, so it is not possible to define the
2300060 // space after the minimal header.
2300061 //
2300062 typedef union
2300063 {
2300064     uint8_t octet[NET_IP_MAX_PACKET_SIZE];
2300065     struct iphdr header;
2300066 } __attribute__((packed)) net_ip_packet_t;
2300067 //
```

```

230068 // Ethernet header.
230069 //
230070 typedef struct
230071 {
230072     uint8_t dst[6];
230073     uint8_t src[6];
230074     uint16_t type;
230075 } __attribute__((packed)) net_ethernet_header_t;
230076 //
230077 // Ethernet frame.
230078 //
230079 // .------.
230080 // | . |
230081 // |-----|
230082 // | .octet[] |
230083 // |-----|
230084 // | .header | .packet |
230085 // |-----+-----|
230086 // | ... | .packet.octet[] |
230087 // |-----+-----|
230088 // | | .packet.header | / / / / / / / / / / / / |
230089 // |-----'-----'-----|
230090 //
230091 typedef union
230092 {
230093     //
230094     uint8_t octet[NET_ETHERNET_MAX_FRAME_SIZE];
230095     //
230096     struct
230097     {
230098         net_ethernet_header_t header;
230099         union
230100         {
230101             //
230102             uint8_t octet[NET_IP_MAX_PACKET_SIZE];
230103             //
230104             struct iphdr header;

```

```
2300105     //
2300106     } __attribute__((packed)) packet;
2300107 } __attribute__((packed));
2300108 //
2300109 } __attribute__((packed)) net_ethernet_frame_t;
2300110 //-----
2300111 //
2300112 // Ethernet buffer.
2300113 //
2300114 typedef struct
2300115 {
2300116     clock_t clock;
2300117     size_t size;
2300118     net_ethernet_frame_t frame;
2300119 } __attribute__((packed)) net_buffer_eth_t;
2300120 //
2300121 // Loopback buffer.
2300122 //
2300123 typedef struct
2300124 {
2300125     clock_t clock;
2300126     size_t size;
2300127     union
2300128     {
2300129         uint8_t octet[NET_MTU];
2300130         struct iphdr header;
2300131     } packet;
2300132 } __attribute__((packed)) net_buffer_lo_t;    // [2]
2300133 //
2300134 // [2] The structure net_ip_packet_t is not used here,
2300135 // because it refers to unfragmented packets, where
2300136 // here the max size might be less.
2300137 //
2300138 //-----
2300139 //
2300140 // Network interfaces table structure
2300141 //
```

```
2300142 typedef struct
2300143 {
2300144     unsigned int type;
2300145     h_addr_t ip; // IPv4 address in host byte order.
2300146     uint8_t m; // Short netmask.
2300147     union
2300148     {
2300149         //
2300150         // Ethernet type data:
2300151         //
2300152         struct
2300153         {
2300154             uint8_t mac[6];
2300155             uintptr_t base_io;
2300156             unsigned char irq;
2300157             net_buffer_eth_t buffer[NET_MAX_BUFFERS];
2300158         } ethernet;
2300159         //
2300160         // Loopback type data:
2300161         //
2300162         struct
2300163         {
2300164             net_buffer_lo_t buffer[NET_MAX_BUFFERS];
2300165         } loopback;
2300166     };
2300167 } net_t;
2300168 //
2300169 // [2] The structure net_ip_packet_t is not used here,
2300170 // because it refers to unfragmented packets, where
2300171 // here the max size might be less.
2300172 //
2300173 extern net_t net_table[NET_MAX_DEVICES];
2300174 //-----
2300175 int net_rx (void);
2300176 int net_tcp (void);
2300177 void net_init (void);
2300178 int net_index (h_addr_t ip);
```

```

2300179 int net_index_eth (h_addr_t ip, uint8_t mac[6],
2300180                 uintptr_t io);
2300181
2300182 net_buffer_eth_t *net_buffer_eth (int n);
2300183 net_buffer_lo_t *net_buffer_lo (int n);
2300184 void net_print (void);
2300185
2300186 //void net_eth_init (int start);
2300187 int net_eth_tx (int dev, void *buffer, size_t size);
2300188 int net_eth_ip_tx (h_addr_t src, h_addr_t dst,
2300189                 const void *packet, size_t size);
2300190 //-----
2300191 #endif

```

94.12.1	kernel/net/arp.h	1552
94.12.2	kernel/net/arp/arp_clean.c	1553
94.12.3	kernel/net/arp/arp_index.c	1554
94.12.4	kernel/net/arp/arp_init.c	1556
94.12.5	kernel/net/arp/arp_print.c	1556
94.12.6	kernel/net/arp/arp_public.c	1557
94.12.7	kernel/net/arp/arp_reference.c	1557
94.12.8	kernel/net/arp/arp_request.c	1558
94.12.9	kernel/net/arp/arp_rx.c	1560
94.12.10	kernel/net/icmp.h	1565
94.12.11	kernel/net/icmp/icmp_rx.c	1566
94.12.12	kernel/net/icmp/icmp_tx.c	1572
94.12.13	kernel/net/icmp/icmp_tx_echo.c	1573

94.12.14	kernel/net/icmp/icmp_tx_unreachable.c	1574
94.12.15	kernel/net/ip.h	1575
94.12.16	kernel/net/ip/ip_checksum.c	1578
94.12.17	kernel/net/ip/ip_header.c	1580
94.12.18	kernel/net/ip/ip_mask.c	1581
94.12.19	kernel/net/ip/ip_public.c	1582
94.12.20	kernel/net/ip/ip_reference.c	1582
94.12.21	kernel/net/ip/ip_rx.c	1583
94.12.22	kernel/net/ip/ip_tx.c	1590
94.12.23	kernel/net/net_buffer_eth.c	1594
94.12.24	kernel/net/net_buffer_lo.c	1595
94.12.25	kernel/net/net_eth_ip_tx.c	1597
94.12.26	kernel/net/net_eth_tx.c	1601
94.12.27	kernel/net/net_index.c	1602
94.12.28	kernel/net/net_index_eth.c	1602
94.12.29	kernel/net/net_init.c	1605
94.12.30	kernel/net/net_print.c	1611
94.12.31	kernel/net/net_public.c	1612
94.12.32	kernel/net/net_rx.c	1612
94.12.33	kernel/net/route.h	1616
94.12.34	kernel/net/route/route_init.c	1617
94.12.35	kernel/net/route/route_print.c	1618

Script e sorgenti del kernel	1551
94.12.36	kernel/net/route/route_public.c 1619
94.12.37	kernel/net/route/route_remote_to_local.c 1619
94.12.38	kernel/net/route/route_remote_to_router.c 1621
94.12.39	kernel/net/route/route_sort.c 1622
94.12.40	kernel/net/tcp.h 1627
94.12.41	kernel/net/tcp/tcp.c 1628
94.12.42	kernel/net/tcp/tcp_close.c 1653
94.12.43	kernel/net/tcp/tcp_connect.c 1656
94.12.44	kernel/net/tcp/tcp_rx_ack.c 1658
94.12.45	kernel/net/tcp/tcp_rx_data.c 1662
94.12.46	kernel/net/tcp/tcp_show.c 1664
94.12.47	kernel/net/tcp/tcp_status.c 1666
94.12.48	kernel/net/tcp/tcp_test.c 1668
94.12.49	kernel/net/tcp/tcp_tx_ack.c 1669
94.12.50	kernel/net/tcp/tcp_tx_raw.c 1671
94.12.51	kernel/net/tcp/tcp_tx_rst.c 1674
94.12.52	kernel/net/tcp/tcp_tx_sock.c 1677
94.12.53	kernel/net/udp.h 1682
94.12.54	kernel/net/udp/udp_tx.c 1683

94.12.1 kernel/net/arp.h



Si veda la sezione [93.1](#).

```
2310001 #ifndef _KERNEL_NET_ARP_H
2310002 #define _KERNEL_NET_ARP_H    1
2310003 //-----
2310004 #include <stdint.h>
2310005 #include <sys/types.h>
2310006 #include <kernel/net/ip.h>
2310007 #include <arpa/inet.h>
2310008 //-----
2310009 #define ARP_HW_ETHERNET    1
2310010 #define ARP_HW_IEEE802    6    // Example, but not
2310011                               // used.
2310012 //-----
2310013 #define ARP_TYPE_REQUEST    1
2310014 #define ARP_TYPE_REPLY    2
2310015 //-----
2310016 typedef struct
2310017 {
2310018     uint16_t hardware_type;
2310019     uint16_t protocol_type;
2310020     uint8_t hardware_address_length;
2310021     uint8_t protocol_address_length;
2310022     uint16_t opcode;
2310023     uint8_t sender_mac[NET_ETHERNET_ADDRESS_LENGTH];
2310024     uint32_t sender_ip;    // Network byte order.
2310025     uint8_t target_mac[NET_ETHERNET_ADDRESS_LENGTH];
2310026     uint32_t target_ip;    // Network byte order.
2310027     uint8_t filler[18];    // [1]
2310028 } __attribute__((packed)) arp_packet_t;
2310029 //
2310030 // [1] The filler is just big enough to get a minimal
2310031 //      Ethernet frame size.
2310032 //
2310033 //-----
2310034 #define ARP_MAX_ITEMS    64
```



```
2310035 #define ARP_MAX_TIME 300 // Seconds.
2310036 //
2310037 typedef struct
2310038 {
2310039     time_t time;
2310040     uint8_t mac[NET_ETHERNET_ADDRESS_LENGTH];
2310041     h_addr_t ip; // Host byte order.
2310042 } arp_t;
2310043 //
2310044 extern arp_t arp_table[ARP_MAX_ITEMS];
2310045 //-----
2310046 void arp_clean (void);
2310047 int arp_index (unsigned char mac[6], h_addr_t ip);
2310048 void arp_init (void);
2310049 void arp_print (void);
2310050 arp_t *arp_reference (void);
2310051 void arp_request (h_addr_t ip);
2310052 int arp_rx (int n, int f);
2310053 //-----
2310054 #endif
```

94.12.2 kernel/net/arp/arp_clean.c

Si veda la sezione [93.1](#).

```
2320001 #include <kernel/net/arp.h>
2320002 #include <kernel/net/ip.h>
2320003 #include <kernel/lib_k.h>
2320004 #include <string.h>
2320005 //-----
2320006 void
2320007 arp_clean (void)
2320008 {
2320009     int a; // ARP table index.
2320010     //
2320011     time_t time_min = k_time (NULL) - ARP_MAX_TIME;
2320012     //
```



```
2320013 //
2320014 //
2320015 for (a = 0; a < ARP_MAX_ITEMS; a++)
2320016 {
2320017     if (arp_table[a].time < time_min)
2320018     {
2320019         //
2320020         // Too old: reset the item to all zeroes.
2320021         //
2320022         memset (&arp_table[a], 0x00,
2320023                 sizeof (arp_table[a]));
2320024     }
2320025 }
2320026 }
```

94.12.3 kernel/net/arp/arp_index.c

<<

Si veda la sezione [93.1](#).

```
2330001 #include <sys/os32.h>
2330002 #include <kernel/net/arp.h>
2330003 #include <kernel/driver/nic/ne2k.h>
2330004 #include <kernel/driver/pci.h>
2330005 #include <kernel/ibm_i386.h>
2330006 #include <errno.h>
2330007 //-----
2330008 extern arp_t arp_table[ARP_MAX_ITEMS];
2330009 //-----
2330010 int
2330011 arp_index (unsigned char mac[6], h_addr_t ip)
2330012 {
2330013     //
2330014     int a;
2330015     //
2330016     // By mac address.
2330017     //
2330018     if (mac != NULL)
```

```
2330019     {
2330020         for (a = 0; a < ARP_MAX_ITEMS; a++)
2330021             {
2330022                 if (arp_table[a].mac[0] == mac[0]
2330023                     && arp_table[a].mac[1] == mac[1]
2330024                     && arp_table[a].mac[2] == mac[2]
2330025                     && arp_table[a].mac[3] == mac[3]
2330026                     && arp_table[a].mac[4] == mac[4]
2330027                     && arp_table[a].mac[5] == mac[5])
2330028                     {
2330029                         return (a);
2330030                     }
2330031             }
2330032     }
2330033     //
2330034     // By IPv4 address.
2330035     //
2330036     if (ip != 0)
2330037         {
2330038             for (a = 0; a < ARP_MAX_ITEMS; a++)
2330039                 {
2330040                     if (arp_table[a].ip == ip)
2330041                         {
2330042                             return (a);
2330043                         }
2330044                 }
2330045         }
2330046     //
2330047     // Not found!
2330048     //
2330049     errset (ENODEV);
2330050     return (-1);
2330051 }
```

94.12.4 kernel/net/arp/arp_init.c



Si veda la sezione [93.1](#).

```
2340001 #include <kernel/net/arp.h>
2340002 #include <string.h>
2340003 //-----
2340004 void
2340005 arp_init (void)
2340006 {
2340007     memset (arp_table, 0x00, sizeof (arp_table));
2340008 }
```

94.12.5 kernel/net/arp/arp_print.c



Si veda la sezione [93.1](#).

```
2350001 #include <kernel/net/arp.h>
2350002 #include <kernel/net/ip.h>
2350003 #include <kernel/lib_k.h>
2350004 //-----
2350005 void
2350006 arp_print (void)
2350007 {
2350008     int a;          // ARP table index.
2350009     //
2350010     for (a = 0; a < ARP_MAX_ITEMS; a++)
2350011     {
2350012         if (arp_table[a].time > 0)
2350013         {
2350014             k_printf ("%i.%i.%i.%i  ",
2350015                 arp_table[a].ip >> 24 & 0x000000FF,
2350016                 arp_table[a].ip >> 16 & 0x000000FF,
2350017                 arp_table[a].ip >> 8 & 0x000000FF,
2350018                 arp_table[a].ip >> 0 & 0x000000FF);
2350019             //
2350020             k_printf ("%02x:%02x:%02x:%02x:%02x:%02x  ",
2350021                 arp_table[a].mac[0],
```

```

2350022         arp_table[a].mac[1],
2350023         arp_table[a].mac[2],
2350024         arp_table[a].mac[3],
2350025         arp_table[a].mac[4],
2350026         arp_table[a].mac[5]);
2350027         //
2350028         k_printf ("%3us\n",
2350029                 (unsigned int)
2350030                 (k_time (NULL) - arp_table[a].time));
2350031         //
2350032     }
2350033 }
2350034 }

```

94.12.6 kernel/net/arp/arp_public.c

<<

Si veda la sezione [93.1](#).

```

2360001 #include <kernel/net/arp.h>
2360002 //-----
2360003 arp_t arp_table[ARP_MAX_ITEMS];
2360004 //-----

```

94.12.7 kernel/net/arp/arp_reference.c

<<

Si veda la sezione [93.1](#).

```

2370001 #include <kernel/net/arp.h>
2370002 #include <kernel/net/ip.h>
2370003 #include <sys/os32.h>
2370004 #include <kernel/lib_k.h>
2370005 //-----
2370006 #define DEBUG 0
2370007 //-----
2370008 arp_t *
2370009 arp_reference (void)
2370010 {

```

```
2370011     int a;           // ARP table index.
2370012     time_t older = 0;
2370013     //
2370014     for (a = 0; a < ARP_MAX_ITEMS; a++)
2370015     {
2370016         if (arp_table[a].time == 0)
2370017         {
2370018             //
2370019             // Enough.
2370020             //
2370021             return (&arp_table[a]);
2370022         }
2370023         //
2370024         older = min (arp_table[a].time, older);
2370025     }
2370026     //
2370027     return (&arp_table[a]);
2370028 }
```

94.12.8 kernel/net/arp/arp_request.c

«

Si veda la sezione [93.1](#).

```
2380001 #include <kernel/net.h>
2380002 #include <kernel/net/arp.h>
2380003 #include <kernel/net/ip.h>
2380004 #include <sys/os32.h>
2380005 #include <kernel/lib_k.h>
2380006 #include <errno.h>
2380007 #include <arpa/inet.h>
2380008 //-----
2380009 #define DEBUG 0
2380010 //-----
2380011 void
2380012 arp_request (h_addr_t ip)
2380013 {
2380014     const uint8_t
```

```
2380015     ethernet_broadcast[NET_ETHERNET_ADDRESS_LENGTH] =
2380016     { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF };
2380017     const uint8_t
2380018     ethernet_null[NET_ETHERNET_ADDRESS_LENGTH] =
2380019     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
2380020     //
2380021     net_ethernet_frame_t frame;
2380022     arp_packet_t *arp = (arp_packet_t *) & frame.packet;
2380023     int n;           // NET table index.
2380024     int i;
2380025     //
2380026     // Send the ARP request to all Ethernet interfaces.
2380027     //
2380028     for (n = 0; n < NET_MAX_DEVICES; n++)
2380029     {
2380030         if (net_table[n].type & NET_DEV_ETH)
2380031         {
2380032             //
2380033             // Build the ARP request packet, starting
2380034             // from Ethernet
2380035             // MAC addresses and protocol type.
2380036             //
2380037             memcpy (frame.header.src,
2380038                     net_table[n].ethernet.mac,
2380039                     (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2380040             memcpy (frame.header.dst, ethernet_broadcast,
2380041                     (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2380042             frame.header.type = htons (NET_PROT_ARP);
2380043             //
2380044             // Now the ARP packet inside.
2380045             //
2380046             arp->hardware_type = htons (ARP_HW_ETHERNET);
2380047             arp->protocol_type = htons (NET_PROT_IP);
2380048             arp->hardware_address_length
2380049                 = NET_ETHERNET_ADDRESS_LENGTH;
2380050             arp->protocol_address_length
2380051                 = NET_IP_ADDRESS_LENGTH;
```

```
2380052     arp->opcode = htons (ARP_TYPE_REQUEST);
2380053     memcpy (arp->sender_mac,
2380054             net_table[n].ethernet.mac,
2380055             (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2380056     arp->sender_ip = htonl (net_table[n].ip);
2380057     memcpy (arp->target_mac, ethernet_null,
2380058             (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2380059     //
2380060     arp->target_ip = htonl (ip);
2380061     //
2380062     for (i = 0; i < (sizeof_array (arp->filler)); i++)
2380063     {
2380064         arp->filler[i] = 0;
2380065     }
2380066     //
2380067     // Send it.
2380068     //
2380069     net_eth_tx (n, &frame,
2380070                 sizeof (arp_packet_t) + 14);
2380071 }
2380072 }
2380073 }
```

94.12.9 kernel/net/arp/arp_rx.c



Si veda la sezione [93.1](#).

```
2390001 #include <kernel/net.h>
2390002 #include <kernel/net/arp.h>
2390003 #include <kernel/net/ip.h>
2390004 #include <sys/os32.h>
2390005 #include <kernel/lib_k.h>
2390006 #include <errno.h>
2390007 #include <arpa/inet.h>
2390008 //-----
2390009 #define DEBUG 0
2390010 //-----
```



```
2390011 int
2390012 arp_rx (int n, int f)
2390013 {
2390014     net_ethernet_frame_t *frame =
2390015         &net_table[n].ethernet.buffer[f].frame;
2390016     arp_packet_t *arp = (arp_packet_t *) & frame->packet;
2390017     int i;
2390018     int a;          // ARP table index.
2390019     arp_t *arp_table_new_item;
2390020     //
2390021     net_ethernet_frame_t ans_frame;
2390022     arp_packet_t *ans_arp =
2390023         (arp_packet_t *) & ans_frame.packet;
2390024     //
2390025     //
2390026     //
2390027     if (n >= NET_MAX_DEVICES || n < 0)
2390028     {
2390029         errset (EINVAL); // Invalid argument.
2390030         return (-1);
2390031     }
2390032     //
2390033     if (!(net_table[n].type & NET_DEV_ETH))
2390034     {
2390035         errset (EINVAL); // Invalid argument.
2390036         return (-1);
2390037     }
2390038     //
2390039     if (ntohs (frame->header.type) != NET_PROT_ARP)
2390040     {
2390041         errset (EINVAL); // Invalid argument.
2390042         return (-1);
2390043     }
2390044     //
2390045     //
2390046     //
2390047     if (ntohs (arp->opcode) == ARP_TYPE_REQUEST)
```

```
2390048     {
2390049         //
2390050         // This is an ARP request: we try to answare if
2390051         // the
2390052         // the IP address is owned.
2390053         //
2390054         if (arp->target_ip == htonl (net_table[n].ip))
2390055             {
2390056                 //
2390057                 // Found IPv4 address. Prepare an answare.
2390058                 //
2390059                 memcpy (ans_frame.header.dst,
2390060                         arp->sender_mac,
2390061                         (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2390062                 memcpy (ans_frame.header.src,
2390063                         net_table[n].ethernet.mac,
2390064                         (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2390065                 ans_frame.header.type = htons (NET_PROT_ARP);
2390066                 ans_arp->hardware_type = htons (ARP_HW_ETHERNET);
2390067                 ans_arp->protocol_type = htons (NET_PROT_IP);
2390068                 ans_arp->hardware_address_length
2390069                     = NET_ETHERNET_ADDRESS_LENGTH;
2390070                 ans_arp->protocol_address_length
2390071                     = NET_IP_ADDRESS_LENGTH;
2390072                 ans_arp->opcode = htons (ARP_TYPE_REPLY);
2390073                 memcpy (ans_arp->sender_mac,
2390074                         net_table[n].ethernet.mac,
2390075                         (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2390076                 ans_arp->sender_ip = htonl (net_table[n].ip);
2390077                 memcpy (ans_arp->target_mac, arp->sender_mac,
2390078                         (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2390079                 ans_arp->target_ip = arp->sender_ip;
2390080                 for (i = 0;
2390081                     i < (sizeof_array (ans_arp->filler)); i++)
2390082                     {
2390083                         ans_arp->filler[i] = 0;
2390084                     }
```

```
2390085         //
2390086         // Send it.
2390087         //
2390088         net_eth_tx (n, &ans_frame,
2390089                   sizeof (arp_packet_t) + 14);
2390090     }
2390091     //
2390092     // Done.
2390093     //
2390094     return (0);
2390095 }
2390096 //
2390097 //
2390098 //
2390099 if (ntohs (arp->opcode) == ARP_TYPE_REPLY)
2390100 {
2390101     //
2390102     // We should save it inside the ARP table.
2390103     //
2390104     for (a = 0; a < ARP_MAX_ITEMS; a++)
2390105     {
2390106         //
2390107         // Check if we already have the same item.
2390108         //
2390109         if (memcmp
2390110             (arp->sender_mac, arp_table[a].mac,
2390111             (size_t) NET_ETHERNET_ADDRESS_LENGTH) == 0)
2390112         {
2390113             if (arp_table[a].ip == ntohl (arp->sender_ip))
2390114             {
2390115                 //
2390116                 // Found: update the time.
2390117                 //
2390118                 arp_table[a].time = k_time (NULL);
2390119                 //
2390120                 // Done.
2390121                 //
```

```
2390122         return (0);
2390123     }
2390124     else
2390125     {
2390126         //
2390127         // There is already the Ethernet
2390128         // address,
2390129         // but the IP is different.
2390130         //
2390131         if (DEBUG)
2390132         {
2390133             k_printf
2390134             ("%s:%i: Ethernet address "
2390135              "%x02:%02:%x02:%x02:%02:%x02 "
2390136              "has more than one IP\n",
2390137              __FILE__, __LINE__,
2390138              arp_table[a].mac[0],
2390139              arp_table[a].mac[1],
2390140              arp_table[a].mac[2],
2390141              arp_table[a].mac[3],
2390142              arp_table[a].mac[4],
2390143              arp_table[a].mac[5]);
2390144             //
2390145             // End of scan.
2390146             //
2390147             break;
2390148         }
2390149     }
2390150 }
2390151 }
2390152 //
2390153 // If we are here, the MAC-IP couple is new: get
2390154 // a new ARP item.
2390155 //
2390156 arp_table_new_item = arp_reference ();
2390157 //
2390158 memcpy (arp_table_new_item->mac, arp->sender_mac,
```

```

2390159         (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2390160     arp_table_new_item->ip = ntohl (arp->sender_ip);
2390161     arp_table_new_item->time = k_time (NULL);
2390162     //
2390163     // Done.
2390164     //
2390165     return (0);
2390166 }
2390167 //
2390168 // If we are here, we don't know the ARP type.
2390169 //
2390170 if (DEBUG)
2390171 {
2390172     k_printf ("%s:%i: unknown ARP type: %i\n",
2390173              (int) ntohs (arp->opcode));
2390174 }
2390175 //
2390176 // Done.
2390177 //
2390178 return (0);
2390179 }

```

94.12.10 kernel/net/icmp.h

Si veda la sezione [93.8](#).

```

2400001 #ifndef _KERNEL_NET_ICMP_H
2400002 #define _KERNEL_NET_ICMP_H      1
2400003 //-----
2400004 #include <stdint.h>
2400005 #include <sys/types.h>
2400006 #include <kernel/net/ip.h>
2400007 #include <netinet/icmp.h>
2400008 //-----
2400009 #define ICMP_HEADER_SIZE      8
2400010 #define ICMP_MAX_PACKET_SIZE  NET_IP_MAX_DATA_SIZE
2400011 #define ICMP_MAX_DATA_SIZE \

```

```
240012         ICMP_MAX_PACKET_SIZE-ICMP_HEADER_SIZE
240013 //-----
240014 //
240015 // ICMP packet, for transmission.
240016 //
240017 typedef struct
240018 {
240019     struct icmphdr header;
240020     uint8_t data[ICMP_MAX_DATA_SIZE];
240021 } __attribute__((packed)) icmp_packet_t;
240022
240023 //-----
240024 int icmp_rx (int i);
240025 //
240026 int icmp_tx (h_addr_t src, h_addr_t dst,
240027             int type, int code, icmp_packet_t * icmp,
240028             size_t size);
240029 //
240030 int icmp_tx_echo (h_addr_t src, h_addr_t dst,
240031                 int type, int code,
240032                 int identifier, int sequence,
240033                 uint8_t * data, size_t size);
240034 //
240035 int icmp_tx_unreachable (h_addr_t src, h_addr_t dst,
240036                        int type, int code,
240037                        uint8_t * data, size_t size);
240038 //
240039 //-----
240040 #endif
```

94.12.11 kernel/net/icmp/icmp_rx.c



Si veda la sezione [93.8](#).

```
2410001 #include <sys/os32.h>
2410002 #include <kernel/lib_k.h>
2410003 #include <arpa/inet.h>
```

```
2410004 #include <kernel/net.h>
2410005 #include <kernel/net/ip.h>
2410006 #include <kernel/net/icmp.h>
2410007 #include <netinet/icmp.h>
2410008 #include <netinet/udp.h>
2410009 #include <kernel/lib_k.h>
2410010 #include <errno.h>
2410011 //-----
2410012 #define DEBUG 0
2410013 //-----
2410014 int
2410015 icmp_rx (int i)
2410016 {
2410017     icmp_packet_t *icmp;
2410018     size_t size;
2410019     struct iphdr *ip;
2410020     struct udphdr *ports;
2410021     h_addr_t dest = 0;
2410022     h_port_t port = 0;
2410023     int s;          // Socket table index.
2410024     //
2410025     //
2410026     //
2410027     if ((i >= IP_MAX_PACKETS) || i < 0)
2410028     {
2410029         errset (EINVAL);
2410030         return (-1);
2410031     }
2410032     //
2410033     // Find the ICMP packet start, and the ICMP packet
2410034     // size (the IP
2410035     // packet size - the IP header size).
2410036     //
2410037     icmp = (icmp_packet_t *) ip_table[i].pdu4;
2410038     size = ntohs (ip_table[i].packet.header.tot_len)
2410039         - (ip_table[i].packet.header.ihl * 4);
2410040     //
```

```
2410041 // This function is used by the kernel, to do
2410042 // something automatically,
2410043 // when some ICMP packets arrive. The kernel does
2410044 // not remove the
2410045 // serviced packets, but must remember what it
2410046 // already have seen.
2410047 // If it finds that the packet clock time stamp is
2410048 // the same as
2410049 // the value saved for the kernel, there is nothing
2410050 // more to be done.
2410051 //
2410052 if (ip_table[i].kernel_serviced == ip_table[i].clock)
2410053 {
2410054     return (0);
2410055 }
2410056 //
2410057 //
2410058 //
2410059 if (icmp->header.type == ICMP_ECHO) // ECHO
2410060 // REQUEST
2410061 {
2410062     size -= sizeof (struct icmphdr);
2410063     //
2410064     // Reply: please note that source and
2410065     // destination IP addresses
2410066     // are now inverted.
2410067     //
2410068     icmp_tx_echo (ntohl
2410069                 (ip_table[i].packet.header.daddr),
2410070                 ntohl (ip_table[i].packet.header.saddr),
2410071                 ICMP_ECHOREPLY, 0,
2410072                 ntohs (icmp->header.un.echo.id),
2410073                 ntohs (icmp->header.un.echo.sequence),
2410074                 icmp->data, size);
2410075     //
2410076     // ICMP echo request are resolved internally,
2410077     // but the packet
```



```
2410078 // might be read from the RAW socket too. So it
2410079 // is not removed
2410080 // from the ip_table[].
2410081 //
2410082 ip_table[i].kernel_serviced = ip_table[i].clock;
2410083 }
2410084 else if (icmp->header.type == ICMP_DEST_UNREACH)
2410085 {
2410086 //
2410087 // UNREACHABLE
2410088 //
2410089 //
2410090 // The code (subtype) is not checked.
2410091 // After the ICMP header there is a copy of the
2410092 // original IP
2410093 // header.
2410094 //
2410095 ip = (struct iphdr *)
2410096 ((uint8_t *) icmp) + sizeof (struct icmphdr);
2410097 //
2410098 dest = ntohl (ip->daddr);
2410099 //
2410100 // If the IP protocol is TCP or UDP, there are
2410101 // also ports.
2410102 //
2410103 if (ip->protocol == IPPROTO_TCP
2410104 || ip->protocol == IPPROTO_UDP)
2410105 {
2410106 //
2410107 // After the IP header copy, there is the
2410108 // TCP or UDP header
2410109 // copy. To be able to get the ports, the
2410110 // UDP and TCP headers
2410111 // are the same.
2410112 //
2410113 ports =
2410114 (struct udphdr *) ((uint8_t *) ip) +
```

```
2410115                                     (ip->ihl * 4));
2410116                                     //
2410117                                     port = ntohs (ports->dest);
2410118                                     //
2410119                                 }
2410120                                 //
2410121                                 // Set corresponding sockets unreachable.
2410122                                 //
2410123                                 for (s = 0; s < SOCK_MAX_SLOTS; s++)
2410124                                 {
2410125                                     if (sock_table[s].active)
2410126                                     {
2410127                                         if (sock_table[s].raddr == dest)
2410128                                         {
2410129                                             if (port == 0
2410130                                                 || port == sock_table[s].rport)
2410131                                             {
2410132                                                 if (icmp->header.code ==
2410133                                                     ICMP_NET_UNREACH)
2410134                                                 {
2410135                                                     sock_table[s].unreach_net = 1;
2410136                                                 }
2410137                                                 else if (icmp->header.code ==
2410138                                                     ICMP_HOST_UNREACH)
2410139                                                 {
2410140                                                     sock_table[s].unreach_host = 1;
2410141                                                 }
2410142                                                 else if (icmp->header.code ==
2410143                                                     ICMP_PROT_UNREACH)
2410144                                                 {
2410145                                                     sock_table[s].unreach_prot = 1;
2410146                                                 }
2410147                                                 else if (icmp->header.code ==
2410148                                                     ICMP_PORT_UNREACH)
2410149                                                 {
2410150                                                     sock_table[s].unreach_port = 1;
2410151                                                 }

```

```
2410152         else
2410153             {
2410154                 sock_table[s].unreach_host = 1;
2410155             }
2410156         }
2410157     }
2410158 }
2410159 }
2410160 }
2410161 else if (icmp->header.type == ICMP_ECHOREPLY)
2410162     {
2410163         //
2410164         // ECHO REPLY
2410165         //
2410166         if (DEBUG)
2410167             {
2410168                 k_printf ("received an echo reply\n");
2410169             }
2410170     }
2410171 else
2410172     {
2410173         //
2410174         // No other ICMP type is handled at the moment,
2410175         // but keep
2410176         // the packet, because might be read from a RAW
2410177         // ICMP socket.
2410178         //
2410179         ;
2410180     }
2410181 //
2410182 return (0);
2410183 }
```

94.12.12 kernel/net/icmp/icmp_tx.c

<<

Si veda la sezione [93.8](#).

```
2420001 #include <sys/os32.h>
2420002 #include <kernel/lib_k.h>
2420003 #include <arpa/inet.h>
2420004 #include <kernel/net.h>
2420005 #include <kernel/net/ip.h>
2420006 #include <kernel/net/icmp.h>
2420007 #include <errno.h>
2420008 //-----
2420009 #define DEBUG 0
2420010 //-----
2420011 int
2420012 icmp_tx (h_addr_t src, h_addr_t dst,
2420013         int type, int code, icmp_packet_t * icmp,
2420014         size_t size)
2420015 {
2420016     uint16_t checksum;
2420017     //
2420018     // Fill the ICMP header.
2420019     //
2420020     icmp->header.type = type;
2420021     icmp->header.code = code;
2420022     icmp->header.checksum = 0;
2420023     //
2420024     // Set the header checksum.
2420025     //
2420026     checksum =
2420027         ~(ip_checksum ((void *) icmp, size, NULL, (size_t) 0));
2420028     icmp->header.checksum = htons (checksum);
2420029     //
2420030     // Send to the lower network level.
2420031     //
2420032     return (ip_tx
2420033             (src, dst, IPPROTO_ICMP, (void *) icmp, size));
2420034 }
```

94.12.13 kernel/net/icmp/icmp_tx_echo.c



Si veda la sezione [93.8](#).

```
2430001 #include <sys/os32.h>
2430002 #include <kernel/lib_k.h>
2430003 #include <arpa/inet.h>
2430004 #include <kernel/net.h>
2430005 #include <kernel/net/ip.h>
2430006 #include <kernel/net/icmp.h>
2430007 #include <errno.h>
2430008 //-----
2430009 #define DEBUG 0
2430010 //-----
2430011 int
2430012 icmp_tx_echo (h_addr_t src, h_addr_t dst, int type,
2430013              int code, int identifier, int sequence,
2430014              uint8_t * data, size_t size)
2430015 {
2430016     icmp_packet_t icmp;
2430017     //
2430018     // Check the data size.
2430019     //
2430020     if (size > ICMP_MAX_DATA_SIZE)
2430021     {
2430022         errset (EINVAL);
2430023         return (-1);
2430024     }
2430025     //
2430026     // Prepare the packet.
2430027     //
2430028     icmp.header.un.echo.id = htons (identifier);
2430029     icmp.header.un.echo.sequence = htons (sequence);
2430030     //
2430031     memcpy (icmp.data, data, size);
2430032     //
2430033     // Send to the lower network level.
2430034     //
```

```
2430035     return (icmp_tx
2430036             (src, dst, type, code, (void *) &icmp,
2430037             (sizeof (struct icmphdr) + size)));
2430038 }
```

94.12.14 kernel/net/icmp/icmp_tx_unreachable.c

<<

Si veda la sezione [93.8](#).

```
2440001 #include <sys/os32.h>
2440002 #include <kernel/lib_k.h>
2440003 #include <arpa/inet.h>
2440004 #include <kernel/net.h>
2440005 #include <kernel/net/ip.h>
2440006 #include <kernel/net/icmp.h>
2440007 #include <errno.h>
2440008 //-----
2440009 #define DEBUG 0
2440010 //-----
2440011 int
2440012 icmp_tx_unreachable (h_addr_t src, h_addr_t dst,
2440013                    int type, int code,
2440014                    uint8_t * data, size_t size)
2440015 {
2440016     icmp_packet_t icmp;
2440017     //
2440018     // Check the data size and reduce if necessary.
2440019     //
2440020     size = min (size, ICMP_MAX_DATA_SIZE);
2440021     //
2440022     if (size < 8)
2440023     {
2440024         //
2440025         // The ICMP destination unreachable data must
2440026         // contain
2440027         // at least 64 bits of the original packet.
2440028         //
```

```

2440029     errset (EINVAL);
2440030     return (-1);
2440031 }
2440032 //
2440033 // Check the type: must be ICMP_DEST_UNREACH.
2440034 //
2440035 if (type != ICMP_DEST_UNREACH)
2440036 {
2440037     errset (EINVAL);
2440038     return (-1);
2440039 }
2440040 //
2440041 // Must reset the «destination unreachable»
2440042 // header.
2440043 //
2440044 memset (&icmp.header.un, 0, (size_t) 4);
2440045 //
2440046 // Copy the data inside the ICMP packet.
2440047 //
2440048 memcpy (icmp.data, data, size);
2440049 //
2440050 // Send to the lower network level.
2440051 //
2440052 return (icmp_tx
2440053         (src, dst, type, code, (void *) &icmp,
2440054         (sizeof (struct icmphdr) + size)));
2440055 }

```

94.12.15 kernel/net/ip.h

Si veda la sezione [93.9](#).

```

2450001 #ifndef _KERNEL_NET_IP_H
2450002 #define _KERNEL_NET_IP_H    1
2450003 //-----
2450004 #include <stdint.h>
2450005 #include <sys/types.h>

```

```
2450006 #include <kernel/net.h>
2450007 #include <netinet/ip.h>
2450008 //-----
2450009 #define IP_VERSION          4
2450010 #define IP_TTL              64
2450011 //-----
2450012 #define IP_MAX_PACKETS     64
2450013 //-----
2450014 //
2450015 // IP packet, for transmission (no options here).
2450016 // It is just the same as 'ip_header_t', with the
2450017 // 'data[]' member addition.
2450018 //
2450019 typedef struct
2450020 {
2450021     struct iphdr header;
2450022     uint8_t data[NET_IP_MAX_DATA_SIZE];
2450023 } __attribute__((packed)) ip_packet_t;
2450024 //
2450025 // IP table for IP packets.
2450026 //
2450027 typedef struct
2450028 {
2450029     clock_t clock;           // [1]
2450030     clock_t kernel_serviced; // [2]
2450031     uint8_t *pdu4;
2450032     union
2450033     {
2450034         uint8_t octet[NET_IP_MAX_PACKET_SIZE];
2450035         struct iphdr header;
2450036     } packet;
2450037 } ip_t;
2450038 //
2450039 // [1] This is the arrival packet clock time stamp.
2450040 //      When a new packet is to be saved inside the
2450041 //      ip_table[], the older packet is replaced, even
2450042 //      if it was not used. No packets are removed,
```



```
2450043 //      because they might be read from a RAW socket,
2450044 //      for some reason.
2450045 //
2450046 // [2] This is only for kernel usage, when a connection
2450047 //      is established by the kernel, without a socket.
2450048 //      The member kernel_serviced is used to remember
2450049 //      to have see a certain packet and that the kernel
2450050 //      does not have to try to service it again. For
2450051 //      example, when an ECHO REQUEST packet arrive, the
2450052 //      kernel answers with an ECHO REPLY, but does not
2450053 //      remove the packet that might be read from a true
2450054 //      RAW socket. So, the kernel writes inside
2450055 //      kernel_serviced the same value found inside
2450056 //      clock, so that it know that it was already read.
2450057 //
2450058 //
2450059 // External IP table data.
2450060 //
2450061 extern ip_t ip_table[IP_MAX_PACKETS];
2450062 //-----
2450063 uint16_t ip_checksum (uint16_t * data1, size_t size1,
2450064                      uint16_t * data2, size_t size2);
2450065 int ip_rx (int n, int f);
2450066 ip_t *ip_reference (void);
2450067 ssize_t ip_header (h_addr_t src, h_addr_t dst,
2450068                  uint16_t id, uint8_t ttl,
2450069                  uint8_t protocol, void *buffer,
2450070                  size_t length);
2450071 int ip_tx (h_addr_t src, h_addr_t dst, int protocol,
2450072           const void *buffer, size_t size);
2450073 h_addr_t ip_mask (int m);
2450074 //-----
2450075 #endif
```

94.12.16 kernel/net/ip/ip_checksum.c



Si veda la sezione [93.9](#).

```
2460001 #include <stdint.h>
2460002 #include <arpa/inet.h>
2460003 #include <kernel/net/ip.h>
2460004 //-----
2460005 uint16_t
2460006 ip_checksum (uint16_t * data1, size_t size1,
2460007              uint16_t * data2, size_t size2)
2460008 {
2460009     int i;
2460010     uint32_t sum;
2460011     uint16_t carry;
2460012     uint16_t checksum;
2460013     uint16_t last;
2460014     uint8_t *octet;
2460015     //
2460016     // 2's complement sum.
2460017     //
2460018     sum = 0;
2460019     //
2460020     if (data1 != NULL)
2460021     {
2460022         for (i = 0; i < (size1 / 2); i++)
2460023         {
2460024             sum += ntohs (data1[i]);
2460025         }
2460026         //
2460027         if (size1 % 2)
2460028         {
2460029             //
2460030             // The size is odd, and the last octet must
2460031             // be accounted too.
2460032             //
2460033             octet = (uint8_t *) data1;
2460034             last = octet[size1 - 1];
```

```
2460035         last = last << 8;
2460036         sum += last;
2460037     }
2460038 }
2460039 if (data2 != NULL)
2460040 {
2460041     for (i = 0; i < (size2 / 2); i++)
2460042     {
2460043         sum += ntohs (data2[i]);
2460044     }
2460045     //
2460046     if (size2 % 2)
2460047     {
2460048         //
2460049         // The size is odd, and the last octet must
2460050         // be accounted too.
2460051         //
2460052         octet = (uint8_t *) data2;
2460053         last = octet[size2 - 1];
2460054         last = last << 8;
2460055         sum += last;
2460056     }
2460057 }
2460058 //
2460059 // Extract the carries and make the checksum.
2460060 //
2460061 carry = sum >> 16;
2460062 checksum = sum & 0x0000FFFF;
2460063 checksum += carry;
2460064 //
2460065 // End of job.
2460066 //
2460067 return (checksum);
2460068 }
```

94.12.17 kernel/net/ip/ip_header.c



Si veda la sezione [93.9](#).

```
2470001 #include <kernel/net.h>
2470002 #include <kernel/net/ip.h>
2470003 #include <sys/os32.h>
2470004 #include <kernel/lib_k.h>
2470005 #include <errno.h>
2470006 #include <arpa/inet.h>
2470007 #include <netinet/ip.h>
2470008 //-----
2470009 #define DEBUG 0
2470010 //-----
2470011 ssize_t
2470012 ip_header (h_addr_t src, h_addr_t dst, uint16_t id,
2470013           uint8_t ttl, uint8_t protocol, void *buffer,
2470014           size_t length)
2470015 {
2470016     struct iphdr header;
2470017     uint16_t checksum;
2470018     //
2470019     // Check size.
2470020     //
2470021     if (length < sizeof (struct iphdr))
2470022     {
2470023         errset (EINVAL);
2470024         return ((ssize_t) - 1);
2470025     }
2470026     //
2470027     // Prepare the header.
2470028     //
2470029     header.version = IP_VERSION;
2470030     header.ihl = sizeof (struct iphdr) / 4;
2470031     header.tos = 0x00;    // Routine, normal.
2470032     header.tot_len = htons (length);
2470033     header.id = htons (id);
2470034     header.frag_off = htons (0x4000);    // Do not
```

```

2470035 // fragment!
2470036 header.ttl = ttl;
2470037 header.protocol = protocol;
2470038 header.check = 0;
2470039 header.saddr = htonl (src);
2470040 header.daddr = htonl (dst);
2470041 //
2470042 // Fix the length.
2470043 //
2470044 length = sizeof (struct iphdr);
2470045 //
2470046 // Now set the header checksum.
2470047 //
2470048 checksum = ~(ip_checksum ((void *) &header, length,
2470049                          NULL, (size_t) 0));
2470050 header.check = htons (checksum);
2470051 //
2470052 // Copy the header to the buffer
2470053 //
2470054 memcpy (buffer, &header, length);
2470055 //
2470056 // Return size written.
2470057 //
2470058 return (length);
2470059 }

```

94.12.18 kernel/net/ip/ip_mask.c

Si veda la sezione [93.9](#).

```

2480001 #include <kernel/net.h>
2480002 #include <kernel/net/route.h>
2480003 #include <kernel/net/arp.h>
2480004 //-----
2480005 h_addr_t
2480006 ip_mask (int m)
2480007 {

```



```
2480008     int i;
2480009     uint32_t mm = 0x80000000;
2480010     h_addr_t mask = 0;
2480011     //
2480012     for (i = 0; i < m; i++)
2480013     {
2480014         mask |= mm;
2480015         mm = mm >> 1;
2480016     }
2480017     //
2480018     return mask;
2480019 }
```

94.12.19 kernel/net/ip/ip_public.c



Si veda la sezione [93.9](#).

```
2490001 #include <kernel/net/ip.h>
2490002 //-----
2490003 ip_t ip_table[IP_MAX_PACKETS];
2490004 //-----
```

94.12.20 kernel/net/ip/ip_reference.c



Si veda la sezione [93.9](#).

```
2500001 #include <kernel/net.h>
2500002 #include <kernel/net/arp.h>
2500003 #include <kernel/lib_k.h>
2500004 #include <kernel/net/ip.h>
2500005 #include <sys/os32.h>
2500006 //-----
2500007 ip_t *
2500008 ip_reference (void)
2500009 {
2500010     int i;           // IP table index.
2500011     clock_t older;
```

```
2500012     int o;           // Older IP table index.
2500013     //
2500014     older = ip_table[0].clock;
2500015     o = 0;
2500016     //
2500017     for (i = 0; i < IP_MAX_PACKETS; i++)
2500018     {
2500019         if (ip_table[i].clock == 0)
2500020         {
2500021             //
2500022             // Enough.
2500023             //
2500024             return (&ip_table[i]);
2500025         }
2500026         //
2500027         if (ip_table[i].clock < older)
2500028         {
2500029             older = ip_table[i].clock;
2500030             o = i;
2500031         }
2500032     }
2500033     //
2500034     return (&ip_table[o]);
2500035 }
```

94.12.21 kernel/net/ip/ip_rx.c

Si veda la sezione [93.9](#).

```
2510001 #include <kernel/net.h>
2510002 #include <kernel/net/arp.h>
2510003 #include <kernel/net/icmp.h>
2510004 #include <kernel/net/ip.h>
2510005 #include <sys/os32.h>
2510006 #include <kernel/lib_k.h>
2510007 #include <errno.h>
2510008 #include <arpa/inet.h>
```



```
2510009 #include <netinet/udp.h>
2510010 //-----
2510011 #define DEBUG 0
2510012 //-----
2510013 int
2510014 ip_rx (int n, int f)
2510015 {
2510016     struct iphdr *header;
2510017     net_ip_packet_t *packet;
2510018     uint16_t checksum;
2510019     size_t size_header;
2510020     ip_t *ip_table_item;
2510021     struct udphdr *udp;
2510022     int i;
2510023     int j;           // Net table index.
2510024     int s;           // Socket table index.
2510025     //
2510026     //
2510027     //
2510028     if (n >= NET_MAX_DEVICES || n < 0)
2510029     {
2510030         errset (EINVAL); // Invalid argument.
2510031         return (-1);
2510032     }
2510033     if (f >= NET_MAX_BUFFERS || f < 0)
2510034     {
2510035         errset (EINVAL); // Invalid argument.
2510036         return (-1);
2510037     }
2510038     //
2510039     // Get the packet link.
2510040     //
2510041     if (net_table[n].type & NET_DEV_LOOP)
2510042     {
2510043         packet = (net_ip_packet_t *)
2510044             & net_table[n].loopback.buffer[f].packet;
2510045     }
```



```
2510046     else if (net_table[n].type & NET_DEV_ETH)
2510047     {
2510048         //
2510049         // It is Ethernet, but must also have an IP
2510050         // packet inside!
2510051         //
2510052         if (ntohs
2510053             (net_table[n].ethernet.buffer[f].frame.
2510054              header.type) != NET_PROT_IP)
2510055         {
2510056             errset (EINVAL);      // Invalid argument.
2510057             return (-1);
2510058         }
2510059         packet = (net_ip_packet_t *)
2510060                 & net_table[n].ethernet.buffer[f].frame.packet;
2510061     }
2510062     else
2510063     {
2510064         errset (EINVAL); // Invalid argument.
2510065         return (-1);
2510066     }
2510067     //
2510068     // The beginning of the packet contains the IP
2510069     // header.
2510070     //
2510071     header = (struct iphdr *) packet;
2510072     //
2510073     // Verify IP header checksum: it is also calculated
2510074     // the real header
2510075     // size.
2510076     //
2510077     size_header = header->ihl * 4;
2510078     checksum =
2510079         ip_checksum ((uint16_t *) header, size_header,
2510080                     NULL, (size_t) 0);
2510081     if (checksum == 0xFFFF || checksum == 0x0000)
2510082     {
```

```
2510083         ; // k_printf ("checksum ok\n");
2510084     }
2510085 else
2510086     {
2510087         k_printf ("BAD CHECKSUM: %04x\n", checksum);
2510088         return (0);
2510089     }
2510090 //
2510091 // Is it a fragment? As we are not able to manage
2510092 // fragments,
2510093 // we just check that it is all zero, ignoring the
2510094 // bit 'DF'.
2510095 // That is why we use a mask 0xBFFF.
2510096 //
2510097 if ((ntohs (header->frag_off) & 0xBFFF) != 0)
2510098     {
2510099         //
2510100         // Sorry, we don't manage fragments.
2510101         //
2510102         k_printf
2510103             ("Sorry: we don't manage IP fragments: %04x\n",
2510104              ntohs (header->frag_off));
2510105         return (0);
2510106     }
2510107 else
2510108     {
2510109         //
2510110         // Find a place inside the IP table.
2510111         //
2510112         ip_table_item = ip_reference ();
2510113         //
2510114         // Copy the packet inside the ip_table[] item,
2510115         // then update
2510116         // the link to the data start inside the packet
2510117         // and the
2510118         // clock_t timestamp.
2510119         //
```

```
2510120     memcpy (ip_table_item->packet.octet, packet,
2510121             ntohs (header->tot_len));
2510122     ip_table_item->pdu4 = ip_table_item->packet.octet;
2510123     ip_table_item->pdu4 += (header->ihl * 4);
2510124     ip_table_item->clock = k_clock ();
2510125 }
2510126 //
2510127 // Check for destination unreachable, scanning the
2510128 // interface table,
2510129 // to see if there is such address here.
2510130 //
2510131 for (j = 0; j < NET_MAX_DEVICES; j++)
2510132 {
2510133     if (net_table[j].ip == ntohl (header->daddr))
2510134     {
2510135         //
2510136         // Found a valid local address.
2510137         //
2510138         break;
2510139     }
2510140 }
2510141 if (j >= NET_MAX_DEVICES)
2510142 {
2510143     //
2510144     // Local address not found: host unreachable,
2510145     // but the packet
2510146     // is taken anyway.
2510147     //
2510148     icmp_tx_unreachable (ntohl (header->daddr),
2510149                         ntohs (header->saddr),
2510150                         ICMP_DEST_UNREACH,
2510151                         ICMP_HOST_UNREACH,
2510152                         packet->octet,
2510153                         ntohs (header->tot_len));
2510154 }
2510155 //
2510156 // Check for port unreachable, scanning the socket
```

```
2510157 // table.
2510158 //
2510159 if (header->protocol == IPPROTO_UDP
2510160     || header->protocol == IPPROTO_TCP)
2510161 {
2510162     //
2510163     // There are ports.
2510164     //
2510165     udp =
2510166         (struct udphdr *) &(packet->octet[header->ihl * 4]);
2510167     //
2510168     for (s = 0; s < SOCK_MAX_SLOTS; s++)
2510169     {
2510170         if (sock_table[s].active
2510171             && sock_table[s].lport == ntohs (udp->dest))
2510172         {
2510173             //
2510174             // Found a matching local port.
2510175             //
2510176             break;
2510177         }
2510178     }
2510179     if (s >= SOCK_MAX_SLOTS)
2510180     {
2510181         //
2510182         // Local port not found: port unreachable,
2510183         // but the packet
2510184         // is taken anyway.
2510185         //
2510186         icmp_tx_unreachable (ntohl (header->daddr),
2510187                             ntohl (header->saddr),
2510188                             ICMP_DEST_UNREACH,
2510189                             ICMP_PORT_UNREACH,
2510190                             packet->octet,
2510191                             ntohs (header->tot_len));
2510192     }
2510193 }
```

```
2510194 //
2510195 // Now do something with the data inside the
2510196 // 'ip_table[]'.
2510197 //
2510198 for (i = 0; i < IP_MAX_PACKETS; i++)
2510199 {
2510200     if (ip_table[i].clock != 0)
2510201     {
2510202         //
2510203         if (ip_table[i].kernel_serviced !=
2510204             ip_table[i].clock
2510205             && ip_table[i].packet.header.protocol ==
2510206             IPPROTO_ICMP)
2510207         {
2510208             icmp_rx (i);
2510209         }
2510210     else
2510211     {
2510212         //
2510213         // At the moment, no other IP protocol
2510214         // managed internally.
2510215         //
2510216         ip_table[i].kernel_serviced =
2510217             ip_table[i].clock;
2510218     }
2510219 }
2510220 }
2510221 //
2510222 //
2510223 //
2510224 return (0);
2510225 }
```

94.12.22 kernel/net/ip/ip_tx.c



Si veda la sezione [93.9](#).

```
2520001 #include <kernel/net.h>
2520002 #include <kernel/net/ip.h>
2520003 #include <kernel/net/route.h>
2520004 #include <sys/os32.h>
2520005 #include <kernel/lib_k.h>
2520006 #include <errno.h>
2520007 #include <arpa/inet.h>
2520008 //-----
2520009 #define DEBUG 0
2520010 //-----
2520011 int
2520012 ip_tx (h_addr_t src, h_addr_t dst, int protocol,
2520013       const void *buffer, size_t size)
2520014 {
2520015     static int id = 0;
2520016     ip_packet_t packet;
2520017     uint16_t checksum;
2520018     int s;           // source net interface.
2520019     int d;           // destination net interface.
2520020     net_buffer_lo_t *loopback;
2520021     //
2520022     // Verify to have a source address.
2520023     //
2520024     if (src == 0)
2520025     {
2520026         //
2520027         // Default source address: get the source
2520028         // address from the routing
2520029         // table, based on the destination.
2520030         //
2520031         src = route_remote_to_local (dst);
2520032         if (src == ((h_addr_t) - 1))
2520033         {
2520034             errset (errno);
```

```
2520035         return (-1);
2520036     }
2520037 }
2520038 //
2520039 // Prepare the packet.
2520040 //
2520041 packet.header.version = IP_VERSION;
2520042 packet.header.ihl = sizeof (struct iphdr) / 4;
2520043 packet.header.tos = 0x0000;    // Routine, normal.
2520044 packet.header.tot_len =
2520045     htons (sizeof (struct iphdr) + size);
2520046 packet.header.id = htons (id++);
2520047 //
2520048 // Do not fragment:
2520049 //
2520050 packet.header.frag_off = htons (0x4000);
2520051 //
2520052 packet.header.ttl = IP_TTL;
2520053 packet.header.protocol = protocol;
2520054 packet.header.check = 0;
2520055 packet.header.saddr = htonl (src);
2520056 packet.header.daddr = htonl (dst);
2520057 //
2520058 // Now set the header checksum.
2520059 //
2520060 checksum =
2520061     ~(ip_checksum
2520062        ((void *) &packet, sizeof (struct iphdr), NULL,
2520063         (size_t) 0));
2520064 packet.header.check = htons (checksum);
2520065 //
2520066 memcpy (packet.data, buffer, size);
2520067 //
2520068 // //////////////////////////////////////
2520069 // Enter here the lower network level.
2520070 // //////////////////////////////////////
2520071 //
```

```
2520072 // The new size includes now the IPv4 header
2520073 //
2520074 size = (sizeof (struct iphdr) + size);
2520075 //
2520076 // Check for PDU size.
2520077 //
2520078 if (size > NET_MTU)
2520079 {
2520080     errset (E_PDU_TOO_BIG);
2520081     return (-1);
2520082 }
2520083 //
2520084 // Find the sender interface.
2520085 //
2520086 s = net_index (src);
2520087 if (s < 0)
2520088 {
2520089     errset (errno); // ENODEV.
2520090     return (-1);
2520091 }
2520092 //
2520093 // Check if the destination is a local interface.
2520094 //
2520095 d = net_index (dst);
2520096 if (d >= 0)
2520097 {
2520098     //
2520099     // It is a local interface, so must change the
2520100     // destination
2520101     // to the loopback device: it must be 'net0'.
2520102     //
2520103     d = 0;
2520104 }
2520105 else
2520106 {
2520107     //
2520108     // Should not be necessary, but for coherence
```



```
2520109         // with the rest
2520110         // of the code...
2520111         //
2520112         d = s;
2520113     }
2520114     //
2520115     // Check if the destination is the loopback
2520116     // interface.
2520117     //
2520118     if (net_table[d].type & NET_DEV_LOOP)
2520119     {
2520120         loopback = net_buffer_lo (d);
2520121         if (loopback == NULL)
2520122         {
2520123             errset (errno);
2520124             return (-1);
2520125         }
2520126         loopback->clock = k_clock ();
2520127         loopback->size = size;
2520128         memcpy (&loopback->packet, (void *) &packet, size);
2520129         return (0);
2520130     }
2520131     //
2520132     // The destination wasn't the loopback interface, so
2520133     // check if the
2520134     // source is an Ethernet device.
2520135     //
2520136     if (net_table[s].type & NET_DEV_ETH)
2520137     {
2520138         //
2520139         // For Ethernet devices another function is
2520140         // responsible
2520141         // for sending the packet.
2520142         //
2520143         return (net_eth_ip_tx
2520144             (src, dst, (void *) &packet, size));
2520145     }
```

```
2520146 //
2520147 // Should never reach the end, but who knows...
2520148 //
2520149 errset (errno); // ENODEV.
2520150 return (-1);
2520151 }
```

94.12.23 kernel/net/net_buffer_eth.c

«

Si veda la sezione [93.17](#).

```
2530001 #include <sys/os32.h>
2530002 #include <kernel/driver/nic/ne2k.h>
2530003 #include <kernel/driver/pci.h>
2530004 #include <kernel/ibm_i386.h>
2530005 #include <errno.h>
2530006 //-----
2530007 net_buffer_eth_t *
2530008 net_buffer_eth (int n)
2530009 {
2530010     int b; // Buffer index.
2530011     int ref = -1; // Reference index.
2530012     clock_t clock = k_clock (); // Reference clock
2530013     // value.
2530014     //
2530015     // Check Ethernet index.
2530016     //
2530017     if ((n > NET_MAX_DEVICES) || (n < 0))
2530018     {
2530019         errset (EINVAL);
2530020         return (NULL);
2530021     }
2530022     //
2530023     if (!(net_table[n].type & NET_DEV_ETH))
2530024     {
2530025         errset (EINVAL);
2530026         return (NULL);
```

```
2530027     }
2530028     //
2530029     // Ethernet found.
2530030     //
2530031     for (b = 0; b < NET_MAX_BUFFERS; b++)
2530032     {
2530033         if (net_table[n].ethernet.buffer[b].clock == 0)
2530034         {
2530035             //
2530036             // Enough.
2530037             //
2530038             return &net_table[n].ethernet.buffer[b];
2530039         }
2530040         else if (net_table[n].ethernet.buffer[b].clock <
2530041                 clock)
2530042         {
2530043             clock = net_table[n].ethernet.buffer[b].clock;
2530044             ref = b;
2530045         }
2530046     }
2530047     //
2530048     // Return the selected frame structure.
2530049     //
2530050     return &net_table[n].ethernet.buffer[ref];
2530051     //
2530052     // Device not found!
2530053     //
2530054     errset (ENODEV);
2530055     return (NULL);
2530056 }
```

94.12.24 kernel/net/net_buffer_io.c

Si veda la sezione [93.17](#).

```
2540001 #include <sys/os32.h>
2540002 #include <kernel/net.h>
```

```
2540003 #include <kernel/driver/nic/ne2k.h>
2540004 #include <kernel/driver/pci.h>
2540005 #include <kernel/ibm_i386.h>
2540006 #include <errno.h>
2540007 //-----
2540008 net_buffer_lo_t *
2540009 net_buffer_lo (int n)
2540010 {
2540011     int b;          // Buffer index.
2540012     int ref = -1;  // Reference index.
2540013     clock_t clock = k_clock (); // Reference clock
2540014     // value.
2540015     //
2540016     // Check NET table index.
2540017     //
2540018     if ((n > NET_MAX_DEVICES) || (n < 0))
2540019     {
2540020         errset (EINVAL);
2540021         return (NULL);
2540022     }
2540023     //
2540024     if (!(net_table[n].type & NET_DEV_LOOP))
2540025     {
2540026         errset (EINVAL);
2540027         return (NULL);
2540028     }
2540029     //
2540030     // Loopback found.
2540031     //
2540032     for (b = 0; b < NET_MAX_BUFFERS; b++)
2540033     {
2540034         if (net_table[n].loopback.buffer[b].clock == 0)
2540035         {
2540036             //
2540037             // Enough.
2540038             //
2540039             return &net_table[n].loopback.buffer[b];
```

```

2540040     }
2540041     else if (net_table[n].loopback.buffer[b].clock <
2540042             clock)
2540043     {
2540044         clock = net_table[n].loopback.buffer[b].clock;
2540045         ref = b;
2540046     }
2540047 }
2540048 //
2540049 // Return the selected frame structure.
2540050 //
2540051 return &net_table[n].loopback.buffer[ref];
2540052 //
2540053 // Device not found!
2540054 //
2540055 errset (ENODEV);
2540056 return (NULL);
2540057 }

```

94.12.25 kernel/net/net_eth_ip_tx.c

Si veda la sezione [93.17](#).

```

2550001 #include <sys/os32.h>
2550002 #include <kernel/net/route.h>
2550003 #include <kernel/net/ip.h>
2550004 #include <kernel/net/arp.h>
2550005 #include <kernel/driver/nic/ne2k.h>
2550006 #include <kernel/driver/pci.h>
2550007 #include <kernel/ibm_i386.h>
2550008 #include <errno.h>
2550009 //-----
2550010 int
2550011 net_eth_ip_tx (h_addr_t src, h_addr_t dst,
2550012               const void *packet, size_t size)
2550013 {
2550014     net_ethernet_frame_t frame;

```

```
2550015 int n; // NET table index.
2550016 int a; // ARP table index.
2550017 int i;
2550018 h_addr_t router;
2550019 //
2550020 // Check for PDU size.
2550021 //
2550022 if (size > NET_ETHERNET_MAX_PACKET_SIZE)
2550023 {
2550024     errset (E_PDU_TOO_BIG);
2550025     return (-1);
2550026 }
2550027 //
2550028 // Find the sender interface address.
2550029 //
2550030 n = net_index_eth (src, NULL, (uintptr_t) 0);
2550031 if (n < 0)
2550032 {
2550033     errset (errno); // ENODEV.
2550034     return (-1);
2550035 }
2550036 //
2550037 // Copy the Ethernet source address into the
2550038 // Ethernet frame
2550039 // header.
2550040 //
2550041 memcpy (frame.header.src, net_table[n].ethernet.mac,
2550042         NET_ETHERNET_ADDRESS_LENGTH);
2550043 //
2550044 // Find if we need a router.
2550045 //
2550046 router = route_remote_to_router (dst);
2550047 //
2550048 if (router != 0 && router != ((h_addr_t) - 1))
2550049 {
2550050     //
2550051     // We need to find the router destination MAC
```

```
2550052      // address.
2550053      //
2550054      a = arp_index (NULL, router);
2550055      if (a < 0)
2550056          {
2550057              //
2550058              // There is not the item inside the ARP
2550059              // table. Send a request
2550060              // and return.
2550061              //
2550062              arp_request (router);
2550063              errset (E_ARP_MISSING);
2550064              return (-1);
2550065          }
2550066      }
2550067  else
2550068      {
2550069          //
2550070          // The destination is inside the local network.
2550071          // Find the destination Ethernet address.
2550072          //
2550073          a = arp_index (NULL, dst);
2550074          if (a < 0)
2550075              {
2550076                  //
2550077                  // There is not the item inside the ARP
2550078                  // table. Send a request
2550079                  // and return.
2550080                  //
2550081                  arp_request (dst);
2550082                  errset (E_ARP_MISSING);
2550083                  return (-1);
2550084              }
2550085      }
2550086      //
2550087      // Copy the Ethernet destination address into the
2550088      // Ethernet frame
```

```
2550089 // header: might be the real destination interface,
2550090 // or the
2550091 // router.
2550092 //
2550093 memcpy (frame.header.dst, arp_table[a].mac,
2550094         NET_ETHERNET_ADDRESS_LENGTH);
2550095 //
2550096 // Set the frame type.
2550097 //
2550098 frame.header.type = htons (NET_PROT_IP);
2550099 //
2550100 // Copy the IP packet.
2550101 //
2550102 memcpy (&frame.packet, packet, size);
2550103 //
2550104 // Fill if the size is too little.
2550105 //
2550106 for (i = size; i < NET_ETHERNET_MIN_PACKET_SIZE; i++)
2550107     {
2550108         frame.packet.octet[i] = 0;
2550109     }
2550110 //
2550111 size = max (size, NET_ETHERNET_MIN_PACKET_SIZE);
2550112 //
2550113 // Now, send the Ethernet frame. Index 'n' is the
2550114 // network
2550115 // device number.
2550116 //
2550117 return (net_eth_tx
2550118         (n, &frame, size + NET_ETHERNET_HEADER_SIZE));
2550119 }
```


94.12.26 kernel/net/net_eth_tx.c



Si veda la sezione [93.17](#).

```
2560001 #include <sys/os32.h>
2560002 #include <kernel/net.h>
2560003 #include <kernel/driver/nic/ne2k.h>
2560004 #include <kernel/driver/pci.h>
2560005 #include <kernel/ibm_i386.h>
2560006 #include <errno.h>
2560007 //-----
2560008 int
2560009 net_eth_tx (int n, void *buffer, size_t size)
2560010 {
2560011     //
2560012     if (n >= NET_MAX_DEVICES || n < 0)
2560013     {
2560014         errset (EINVAL);
2560015         return (-1);
2560016     }
2560017     if (!(net_table[n].type & NET_DEV_ETH))
2560018     {
2560019         errset (EINVAL);
2560020         return (-1);
2560021     }
2560022     //
2560023     if (net_table[n].type == NET_DEV_ETH_NE2K)
2560024     {
2560025         return (ne2k_tx
2560026                 (net_table[n].ethernet.base_io, buffer,
2560027                 size));
2560028     }
2560029     //
2560030     // If we are here, there is not the driver for the
2560031     // Ethernet device.
2560032     //
2560033     errset (ENODEV);
2560034     return (-1);
```

```
2560035 }
```

94.12.27 kernel/net/net_index.c

<<

Si veda la sezione [93.17](#).

```
2570001 #include <sys/os32.h>
2570002 #include <kernel/net.h>
2570003 #include <errno.h>
2570004 #include <stdint.h>
2570005 #include <arpa/inet.h>
2570006 //-----
2570007 int
2570008 net_index (h_addr_t ip)
2570009 {
2570010     //
2570011     int n;
2570012     //
2570013     // By IPv4 address.
2570014     //
2570015     if (ip != 0)
2570016     {
2570017         for (n = 0; n < NET_MAX_DEVICES; n++)
2570018         {
2570019             if (net_table[n].ip == ip)
2570020             {
2570021                 return (n);
2570022             }
2570023         }
2570024     }
2570025     //
2570026     // Not found!
2570027     //
2570028     errset (ENODEV);
2570029     return (-1);
2570030 }
```

94.12.28 kernel/net/net_index_eth.c



Si veda la sezione [93.17](#).

```
2580001 #include <sys/os32.h>
2580002 #include <kernel/net.h>
2580003 #include <errno.h>
2580004 #include <stdint.h>
2580005 #include <arpa/inet.h>
2580006 //-----
2580007 int
2580008 net_index_eth (h_addr_t ip, uint8_t mac[6], uintptr_t io)
2580009 {
2580010     //
2580011     int n;
2580012     //
2580013     // If 'ip' is not zero, then find the Ethernet table
2580014     // index by that
2580015     // value.
2580016     //
2580017     if (ip != 0)
2580018     {
2580019         for (n = 0; n < NET_MAX_DEVICES; n++)
2580020         {
2580021             if (net_table[n].type & NET_DEV_ETH)
2580022             {
2580023                 if (net_table[n].ip == ip)
2580024                 {
2580025                     return (n);
2580026                 }
2580027             }
2580028         }
2580029     }
2580030     //
2580031     // By mac address.
2580032     //
2580033     if (mac != NULL)
2580034     {
```

```
2580035     for (n = 0; n < NET_MAX_DEVICES; n++)
2580036     {
2580037         if (net_table[n].type & NET_DEV_ETH)
2580038         {
2580039             if (net_table[n].ethernet.mac[0] ==
2580040                 mac[0]
2580041                 && net_table[n].ethernet.mac[1] ==
2580042                 mac[1]
2580043                 && net_table[n].ethernet.mac[2] ==
2580044                 mac[2]
2580045                 && net_table[n].ethernet.mac[3] ==
2580046                 mac[3]
2580047                 && net_table[n].ethernet.mac[4] ==
2580048                 mac[4]
2580049                 && net_table[n].ethernet.mac[5] == mac[5])
2580050             {
2580051                 return (n);
2580052             }
2580053         }
2580054     }
2580055 }
2580056 //
2580057 // By hardware I/O address.
2580058 //
2580059 if (io > 0)
2580060 {
2580061     for (n = 0; n < NET_MAX_DEVICES; n++)
2580062     {
2580063         if (net_table[n].type & NET_DEV_ETH)
2580064         {
2580065             if (net_table[n].ethernet.base_io == io)
2580066             {
2580067                 return (n);
2580068             }
2580069         }
2580070     }
2580071 }
```

```
2580072 //
2580073 // Not found!
2580074 //
2580075 errset (ENODEV);
2580076 return (-1);
2580077 }
```

94.12.29 kernel/net/net_init.c

Si veda la sezione [93.17](#).

```
2590001 #include <kernel/net.h>
2590002 #include <kernel/net/route.h>
2590003 #include <kernel/net/arp.h>
2590004 #include <kernel/lib_s.h>
2590005 #include <kernel/proc.h>
2590006 #include <kernel/multiboot.h>
2590007 #include <stdlib.h>
2590008 #include <kernel/lib_k.h>
2590009 #include <string.h>
2590010 #include <errno.h>
2590011 #include <kernel/driver/pci.h>
2590012 #include <kernel/driver/nic/ne2k.h>
2590013 //-----
2590014 static void net_eth_init (int start);
2590015 //-----
2590016 void
2590017 net_init (void)
2590018 {
2590019     int n;           // NET device table index.
2590020     int b;           // Buffer NET device index.
2590021     int i;
2590022     char *net = "net0";
2590023     char *route = "route0";
2590024     char **argument;
2590025     in_addr_t ip_a;
2590026     in_addr_t ip_b;
```



```
2590027     int status;
2590028     //
2590029     // Reset the NET device table.
2590030     //
2590031     for (n = 0; n < NET_MAX_DEVICES; n++)
2590032     {
2590033         net_table[n].type = NET_DEV_NULL;
2590034     }
2590035     //
2590036     // Set up the loopback interface.
2590037     //
2590038     net_table[0].type = NET_DEV_LOOPBACK;
2590039     net_table[0].ip = INADDR_LOOPBACK;
2590040     net_table[0].m = 8;
2590041     //
2590042     for (b = 0; b < NET_MAX_BUFFERS; b++)
2590043     {
2590044         net_table[0].loopback.buffer[b].clock = 0;
2590045     }
2590046     //
2590047     // Prepare ARP table
2590048     //
2590049     arp_init ();
2590050     //
2590051     // Add Ethernet devices, but starting from the
2590052     // second interface
2590053     // inside the net_table[].
2590054     //
2590055     net_eth_init (1);
2590056     //
2590057     // Prepare routes.
2590058     //
2590059     route_init ();
2590060     route_sort ();
2590061     //
2590062     // Command line options: counter 'i' is scanned like
2590063     // a character.
```

```
2590064 //
2590065 for (i = '0'; i <= '9'; i++)
2590066 {
2590067     net[3] = i;
2590068     argument = mboot_cmdline_opt (net, ",");
2590069     if (argument != NULL)
2590070     {
2590071         //
2590072         status = inet_pton (AF_INET, argument[2], &ip_a);
2590073         if (status != 1)
2590074         {
2590075             continue;
2590076         }
2590077         //
2590078         s_ipconfig ((pid_t) 0, atoi (argument[1]),
2590079                     ip_a, atoi (argument[3]));
2590080     }
2590081 }
2590082 //
2590083 for (i = '0'; i <= '9'; i++)
2590084 {
2590085     route[5] = i;
2590086     argument = mboot_cmdline_opt (route, ",");
2590087     if (argument != NULL)
2590088     {
2590089         //
2590090         status = inet_pton (AF_INET, argument[1], &ip_a);
2590091         if (status != 1)
2590092         {
2590093             continue;
2590094         }
2590095         status = inet_pton (AF_INET, argument[3], &ip_b);
2590096         if (status != 1)
2590097         {
2590098             continue;
2590099         }
2590100         //
```

```
2590101         s_routeadd ((pid_t) 0,
2590102                     ip_a, atoi (argument[2]),
2590103                     ip_b, atoi (argument[4]));
2590104     }
2590105 }
2590106 //
2590107 //
2590108 //
2590109 net_print ();
2590110 route_print ();
2590111 }
2590112
2590113 //-----
2590114 static void
2590115 net_eth_init (int start)
2590116 {
2590117     int p;          // PCI table index.
2590118     int n;          // NET devices table index.
2590119     int i;
2590120     int j;
2590121     //
2590122     static const struct
2590123     {
2590124         unsigned short vendor;
2590125         unsigned short device;
2590126     } type_ne2k[] =
2590127     {
2590128         {
2590129             0x10ec, 0x8029},      // RealTek_RTL_8029
2590130         {
2590131             0x1050, 0x0940},      // Winbond_89C940
2590132         {
2590133             0x11f6, 0x1401},      // Compex_RL2000
2590134         {
2590135             0x8e2e, 0x3000},      // KTI_ET32P2
2590136         {
2590137             0x4a14, 0x5000},      // NetVin_NV5000SC
```



```
2590138     {
2590139     0x1106, 0x0926},    // Via_86C926
2590140     {
2590141     0x10bd, 0x0e34},    // SureCom_NE34
2590142     {
2590143     0x1050, 0x5a5a},    // Winbond_W89C940F
2590144     {
2590145     0x12c3, 0x0058},    // Holtek_HT80232
2590146     {
2590147     0x12c3, 0x5598},    // Holtek_HT80229
2590148     {
2590149     0x8c4a, 0x1980},    // Winbond_89C940_8c4a
2590150 };
2590151 //
2590152 //
2590153 //
2590154 n = start;
2590155 //
2590156 // Scan the PCI table and find NE2K Ethernet
2590157 // devices.
2590158 //
2590159 for (p = 0;
2590160      p < PCI_MAX_DEVICES && n < NET_MAX_DEVICES; p++)
2590161     {
2590162     for (i = 0; i < sizeof_array (type_ne2k); i++)
2590163     {
2590164     if (pci_table[p].vendor_id ==
2590165         type_ne2k[i].vendor
2590166         && pci_table[p].device_id ==
2590167         type_ne2k[i].device)
2590168     {
2590169         //
2590170         // Verify if the NIC is really a NE2K.
2590171         //
2590172         if (ne2k_check (pci_table[p].base_io) == 0)
2590173             {
2590174                 //
```

```
2590175 // Reset the NIC and get the
2590176 // physical address.
2590177 //
2590178 if (ne2k_reset (pci_table[p].base_io,
2590179               net_table[n].
2590180               ethernet.mac) == 0)
2590181     {
2590182         //
2590183         // New.
2590184         //
2590185         net_table[n].type = NET_DEV_ETH_NE2K;
2590186         net_table[n].ethernet.base_io =
2590187             pci_table[p].base_io;
2590188         net_table[n].ethernet.irq =
2590189             pci_table[p].irq;
2590190         //
2590191         for (j = 0; j < NET_MAX_BUFFERS; j++)
2590192             {
2590193                 net_table[n].ethernet.
2590194                     buffer[j].clock = 0;
2590195             }
2590196         //
2590197         // Go to next NET table element.
2590198         //
2590199         n++;
2590200         //
2590201         break;
2590202     }
2590203 }
2590204 }
2590205 }
2590206 }
2590207 }
```

94.12.30 kernel/net/net_print.c



Si veda la sezione [93.17](#).

```
2600001 #include <sys/os32.h>
2600002 #include <kernel/net.h>
2600003 #include <errno.h>
2600004 //-----
2600005 void
2600006 net_print (void)
2600007 {
2600008     int n;          // NET devices table index.
2600009     char string[80];
2600010     //
2600011     //
2600012     //
2600013     k_printf ("dev      "
2600014              "address/mask      "
2600015              "mac                " "io      irq\n");
2600016     //
2600017     for (n = 0; n < NET_MAX_DEVICES; n++)
2600018     {
2600019         if (net_table[n].type != NET_DEV_NULL)
2600020         {
2600021             sprintf (string, "net%i      ", n);
2600022             string[6] = '\0';
2600023             k_printf ("%s", string);
2600024             //
2600025             sprintf (string, "%i.%i.%i.%i/%i "
2600026                     "                ",
2600027                     net_table[n].ip >> 24 & 0x000000FF,
2600028                     net_table[n].ip >> 16 & 0x000000FF,
2600029                     net_table[n].ip >> 8 & 0x000000FF,
2600030                     net_table[n].ip >> 0 & 0x000000FF,
2600031                     net_table[n].m);
2600032             string[20] = '\0';
2600033             k_printf ("%s", string);
2600034             //
```

```

2600035         if (net_table[n].type & NET_DEV_ETH)
2600036             {
2600037                 k_printf
2600038                 ("%02x:%02x:%02x:%02x:%02x:%02x  "
2600039                 "0x%04x  %i",
2600040                 net_table[n].ethernet.mac[0],
2600041                 net_table[n].ethernet.mac[1],
2600042                 net_table[n].ethernet.mac[2],
2600043                 net_table[n].ethernet.mac[3],
2600044                 net_table[n].ethernet.mac[4],
2600045                 net_table[n].ethernet.mac[5],
2600046                 net_table[n].ethernet.base_io,
2600047                 net_table[n].ethernet.irq);
2600048             }
2600049         k_printf ("\n");
2600050     }
2600051 }
2600052 }

```

94.12.31 kernel/net/net_public.c

«

Si veda la sezione [93.17](#).

```

2610001 #include <kernel/net.h>
2610002 //-----
2610003 net_t net_table[NET_MAX_DEVICES];
2610004 //-----

```

94.12.32 kernel/net/net_rx.c

«

Si veda la sezione [93.17](#).

```

2620001 #include <arpa/inet.h>
2620002 #include <kernel/net.h>
2620003 #include <kernel/net/arp.h>
2620004 #include <sys/os32.h>
2620005 #include <kernel/lib_k.h>

```

```
2620006 //-----
2620007 #define DEBUG 0
2620008 //-----
2620009 int
2620010 net_rx (void)
2620011 {
2620012     int n;          // NET table index.
2620013     int b;          // Frame index.
2620014     net_ethernet_frame_t *frame;
2620015     int counter = 0;
2620016     //
2620017     // Scan NET table.
2620018     //
2620019     for (n = 0; n < NET_MAX_DEVICES; n++)
2620020     {
2620021         //
2620022         // Ethernet.
2620023         //
2620024         if (net_table[n].type & NET_DEV_ETH)
2620025         {
2620026             for (b = 0; b < NET_MAX_BUFFERS; b++)
2620027             {
2620028                 if (net_table[n].ethernet.buffer[b].clock > 0)
2620029                 {
2620030                     frame = (net_ethernet_frame_t *)
2620031                         & net_table[n].ethernet.buffer[b].frame;
2620032                     //
2620033                     if (ntohs (frame->header.type) ==
2620034                         NET_PROT_ARP)
2620035                     {
2620036                         arp_rx (n, b);
2620037                         //
2620038                         // Remove packet from buffer.
2620039                         //
2620040                         net_table[n].ethernet.buffer[b].
2620041                             clock = 0;
2620042                         //

```

```
2620043         // Increment the packet received
2620044         // counter.
2620045         //
2620046         counter++;
2620047     }
2620048     else if (ntohs (frame->header.type)
2620049             == NET_PROT_IP)
2620050     {
2620051         ip_rx (n, b);
2620052         //
2620053         // Remove packet from buffer.
2620054         //
2620055         net_table[n].ethernet.buffer[b].
2620056             clock = 0;
2620057         //
2620058         // Increment the packet received
2620059         // counter.
2620060         //
2620061         counter++;
2620062     }
2620063     else
2620064     {
2620065         //
2620066         // Unknown frame type.
2620067         //
2620068         k_printf
2620069             ("received an unknown frame "
2620070             "type %04x\n",
2620071             ntohs (frame->header.type));
2620072         //
2620073         // Remove packet from buffer.
2620074         //
2620075         net_table[n].ethernet.buffer[b].
2620076             clock = 0;
2620077         //
2620078         // Increment the packet received
2620079         // counter anyway.
```

```
2620080                                     //
2620081                                     counter++;
2620082                                     }
2620083                                 }
2620084                            }
2620085                        }
2620086                    //
2620087                    // Loopback
2620088                    //
2620089                    else if (net_table[n].type & NET_DEV_LOOP)
2620090                    {
2620091                        for (b = 0; b < NET_MAX_BUFFERS; b++)
2620092                        {
2620093                            if (net_table[n].loopback.buffer[b].clock > 0)
2620094                            {
2620095                                ip_rx (n, b);
2620096                                //
2620097                                // Remove packet from buffer.
2620098                                //
2620099                                net_table[n].loopback.buffer[b].clock = 0;
2620100                                //
2620101                                // Increment the packet received
2620102                                // counter.
2620103                                //
2620104                                counter++;
2620105                                //
2620106                            }
2620107                        }
2620108                    }
2620109                }
2620110            //
2620111            // Remove ARP items that are too old.
2620112            //
2620113            arp_clean ();
2620114            //
2620115            //
2620116            //
```

```
2620117     return (counter);
2620118 }
```

94.12.33 kernel/net/route.h

<<

Si veda la sezione [93.21](#).

```
2630001 #ifndef _KERNEL_NET_ROUTE_H
2630002 #define _KERNEL_NET_ROUTE_H    1
2630003 //-----
2630004 #include <stdint.h>
2630005 #include <sys/types.h>
2630006 #include <kernel/net.h>
2630007 #include <netinet/in.h>
2630008 //-----
2630009 #define ROUTE_MAX_ROUTES 16
2630010 //
2630011 // Route table element.
2630012 //
2630013 typedef struct
2630014 {
2630015     h_addr_t network;    // 32 bit, host byte order.
2630016     h_addr_t netmask;   // 32 bit, host byte order.
2630017     h_addr_t router;    // 32 bit, host byte order.
2630018     uint8_t m;         // Short netmask.
2630019     uint8_t interface;
2630020 } route_t;
2630021 //
2630022 // External routing table data.
2630023 //
2630024 extern route_t route_table[ROUTE_MAX_ROUTES];
2630025 //-----
2630026 void route_init (void);
2630027 void route_sort (void);
2630028 void route_print (void);
2630029 h_addr_t route_remote_to_local (h_addr_t remote);
2630030 h_addr_t route_remote_to_router (h_addr_t remote);
```



```
2630031
2630032 //-----
2630033 #endif
```

94.12.34 kernel/net/route/route_init.c



Si veda la sezione [93.21](#).

```
2640001 #include <arpa/inet.h>
2640002 #include <sys/os32.h>
2640003 #include <kernel/net/route.h>
2640004 #include <errno.h>
2640005 #include <netinet/in.h>
2640006 //-----
2640007 void
2640008 route_init (void)
2640009 {
2640010     //
2640011     // Reset the table with 0xFF.
2640012     //
2640013     memset (route_table, 0xFF, sizeof (route_table));
2640014     //
2640015     // Put the loopback routing.
2640016     //
2640017     route_table[0].netmask = 0xFF000000; // Little
2640018     // endian.
2640019     route_table[0].m = 8;
2640020     route_table[0].network =
2640021         INADDR_LOOPBACK & route_table[0].netmask;
2640022     route_table[0].router = 0;
2640023     route_table[0].interface = 0;
2640024 }
```

94.12.35 kernel/net/route/route_print.c



Si veda la sezione [93.21](#).

```
2650001 #include <arpa/inet.h>
2650002 #include <sys/os32.h>
2650003 #include <kernel/net/route.h>
2650004 #include <kernel/lib_k.h>
2650005 #include <errno.h>
2650006 //-----
2650007 void
2650008 route_print (void)
2650009 {
2650010     int r;          // Routing table index.
2650011     char string[80];
2650012     //
2650013     k_printf ("Destination/mask      "
2650014             "Router                  " "Interface\n");
2650015     //
2650016     for (r = 0; r < ROUTE_MAX_ROUTES; r++)
2650017     {
2650018         if (route_table[r].network == 0xFFFFFFFF)
2650019         {
2650020             //
2650021             // Empty item.
2650022             //
2650023             continue;
2650024         }
2650025         //
2650026         sprintf (string, "%i.%i.%i.%i/%i"
2650027                 "                ",
2650028                 route_table[r].network >> 24 & 0x000000FF,
2650029                 route_table[r].network >> 16 & 0x000000FF,
2650030                 route_table[r].network >> 8 & 0x000000FF,
2650031                 route_table[r].network >> 0 & 0x000000FF,
2650032                 route_table[r].m);
2650033         string[19] = '\\0';
2650034         k_printf ("%s", string);
```

```

2650035 //
2650036 if (route_table[r].router == 0)
2650037 {
2650038     k_printf ("                ");
2650039 }
2650040 else
2650041 {
2650042     sprintf (string, "%i.%i.%i.%i"
2650043             "                ",
2650044             route_table[r].router >> 24 & 0x000000FF,
2650045             route_table[r].router >> 16 & 0x000000FF,
2650046             route_table[r].router >> 8 & 0x000000FF,
2650047             route_table[r].router >> 0 & 0x000000FF);
2650048     string[16] = '\0';
2650049     k_printf ("%s", string);
2650050 }
2650051 //
2650052 k_printf ("net%i\n", route_table[r].interface);
2650053 }
2650054 }

```

94.12.36 kernel/net/route/route_public.c



Si veda la sezione [93.21](#).

```

2660001 #include <kernel/net/route.h>
2660002 //-----
2660003 route_t route_table[ROUTE_MAX_ROUTES];
2660004 //-----

```

94.12.37 kernel/net/route/route_remote_to_local.c



Si veda la sezione [93.21](#).

```

2670001 #include <arpa/inet.h>
2670002 #include <sys/os32.h>
2670003 #include <kernel/net/route.h>

```

```
2670004 #include <kernel/lib_k.h>
2670005 #include <errno.h>
2670006 //-----
2670007 h_addr_t
2670008 route_remote_to_local (h_addr_t remote)
2670009 {
2670010     int r;           // Routing table index.
2670011     int d;           // Network interface number.
2670012     h_addr_t network;
2670013     //
2670014     for (r = 0; r < ROUTE_MAX_ROUTES; r++)
2670015     {
2670016         //
2670017         // Calculate the remote network address based on
2670018         // the current
2670019         // router item netmask.
2670020         //
2670021         network = remote & route_table[r].netmask;
2670022         //
2670023         // Compare the calculated network address with
2670024         // the remote
2670025         // network.
2670026         //
2670027         if (route_table[r].network == network)
2670028         {
2670029             //
2670030             // Found.
2670031             //
2670032             d = route_table[r].interface;
2670033             //
2670034             // Check inside the network interfaces.
2670035             //
2670036             if (net_table[d].ip == 0)
2670037             {
2670038                 errset (ENODEV);
2670039                 return ((h_addr_t) - 1);
2670040             }
```

```
2670041         else
2670042         {
2670043             return (net_table[d].ip);
2670044         }
2670045     }
2670046 }
2670047 //
2670048 // Sorry: destination not found.
2670049 //
2670050 errset (EADDRNOTAVAIL);
2670051 return ((h_addr_t) - 1);
2670052 }
```

94.12.38 kernel/net/route/route_remote_to_router.c



Si veda la sezione [93.21](#).

```
2680001 #include <arpa/inet.h>
2680002 #include <sys/os32.h>
2680003 #include <kernel/net/route.h>
2680004 #include <kernel/lib_k.h>
2680005 #include <errno.h>
2680006 //-----
2680007 h_addr_t
2680008 route_remote_to_router (h_addr_t remote)
2680009 {
2680010     int r;          // Routing table index.
2680011     h_addr_t network;
2680012     //
2680013     for (r = 0; r < ROUTE_MAX_ROUTES; r++)
2680014     {
2680015         //
2680016         // Calculate the remote network address based on
2680017         // the current
2680018         // router item netmask.
2680019         //
2680020         network = remote & route_table[r].netmask;
```



```
2690012 //
2690013 static uint8_t swap[sizeof (route_t)];
2690014 static int comp (void *a, void *b);
2690015 //-----
2690016 void
2690017 route_sort (void)
2690018 {
2690019     qsort (route_table, ROUTE_MAX_ROUTES,
2690020           sizeof (route_t), comp);
2690021 }
2690022
2690023 //-----
2690024 static int
2690025 comp (void *a, void *b)
2690026 {
2690027     route_t *route_a = a;
2690028     route_t *route_b = b;
2690029     uint8_t m_a = route_a->m;
2690030     uint8_t m_b = route_b->m;
2690031     //
2690032     if (m_a > m_b)
2690033         return (-1);
2690034     if (m_a < m_b)
2690035         return (1);
2690036     return (0);
2690037 }
2690038
2690039 //-----
2690040 static void
2690041 qsort (void *base, size_t nmemb, size_t size,
2690042        int (*compare) (void *, void *))
2690043 {
2690044     if (size <= 1)
2690045     {
2690046         //
2690047         // There is nothing to sort!
2690048         //
```

```
2690049     return;
2690050     }
2690051     else
2690052     {
2690053         sort ((char *) base, size, 0, (int) (nmemb - 1),
2690054             compare);
2690055     }
2690056 }
2690057
2690058 //-----
2690059 static void
2690060 sort (char *array, size_t size, int a, int z,
2690061      int (*compare) (void *, void *))
2690062 {
2690063     int loc;
2690064     //
2690065     if (z > a)
2690066     {
2690067         loc = part (array, size, a, z, compare);
2690068         if (loc >= 0)
2690069         {
2690070             sort (array, size, a, loc - 1, compare);
2690071             sort (array, size, loc + 1, z, compare);
2690072         }
2690073     }
2690074 }
2690075
2690076 //-----
2690077 static int
2690078 part (char *array, size_t size, int a, int z,
2690079      int (*compare) (void *, void *))
2690080 {
2690081     int i;
2690082     int loc;
2690083     //
2690084     if (z <= a)
2690085     {
```



```
2690086     errset (EUNKNOWN);           // Should never
2690087     // happen.
2690088     return (-1);
2690089 }
2690090 //
2690091 // Index 'i' after the first element; index 'loc' at
2690092 // the last
2690093 // position.
2690094 //
2690095 i = a + 1;
2690096 loc = z;
2690097 //
2690098 // Loop as long as index 'loc' is higher than index
2690099 // 'i'.
2690100 // When index 'loc' is less or equal to index 'i',
2690101 // then, index 'loc' is the right position for the
2690102 // first element of the current piece of array.
2690103 //
2690104 for (;;)
2690105 {
2690106     //
2690107     // Index 'i' goes up...
2690108     //
2690109     for (; i < loc; i++)
2690110     {
2690111         if (compare
2690112             (&array[i * size], &array[a * size]) > 0)
2690113         {
2690114             break;
2690115         }
2690116     }
2690117     //
2690118     // Index 'loc' goes down...
2690119     //
2690120     for (; loc--;)
2690121     {
2690122         if (compare
```

```
2690123         (&array[loc * size], &array[a * size]) <= 0)
2690124     {
2690125         break;
2690126     }
2690127 }
2690128 //
2690129 // Swap elements related to index 'i' and 'loc'.
2690130 //
2690131 if (loc <= i)
2690132     {
2690133         //
2690134         // The array is completely scanned.
2690135         //
2690136         break;
2690137     }
2690138 else
2690139     {
2690140         memcpy (swap, &array[loc * size], size);
2690141         memcpy (&array[loc * size], &array[i * size],
2690142             size);
2690143         memcpy (&array[i * size], swap, size);
2690144     }
2690145 }
2690146 //
2690147 // Swap the first element with the one related to
2690148 // the
2690149 // index 'loc'.
2690150 //
2690151 memcpy (swap, &array[loc * size], size);
2690152 memcpy (&array[loc * size], &array[a * size], size);
2690153 memcpy (&array[a * size], swap, size);
2690154 //
2690155 // Return the index 'loc'.
2690156 //
2690157 return (loc);
2690158 }
```

94.12.40 kernel/net/tcp.h



Si veda la sezione [93.23](#).

```
2700001 #ifndef _KERNEL_NET_TCP_H
2700002 #define _KERNEL_NET_TCP_H    1
2700003 //-----
2700004 #include <netinet/tcp.h>
2700005 #include <kernel/net.h>
2700006 //-----
2700007 #define TCP_HEADER_SIZE      20
2700008 #define TCP_MAX_PACKET_SIZE  NET_IP_MAX_DATA_SIZE
2700009 #define TCP_MAX_DATA_SIZE    \
2700010     TCP_MAX_PACKET_SIZE-TCP_HEADER_SIZE
2700011 //
2700012 #define TCP_MAX_DELAY        (CLOCKS_PER_SEC*2)
2700013 //-----
2700014 //
2700015 // TCP packet, for transmission.
2700016 //
2700017 typedef struct
2700018 {
2700019     struct tcphdr header;
2700020     uint8_t data[TCP_MAX_DATA_SIZE];
2700021 } __attribute__((packed)) tcp_packet_t;
2700022 //
2700023 // TCP pseudo header for checksum calculation.
2700024 //
2700025 typedef struct
2700026 {
2700027     in_addr_t saddr;
2700028     in_addr_t daddr;
2700029     uint8_t zero;
2700030     uint8_t protocol;
2700031     uint16_t length;
2700032 } __attribute__((packed)) tcp_pseudo_header_t;
2700033 //-----
2700034 #define TCP_FLAG_NULL    0
```

```

2700035 #define TCP_FLAG_ACK 1
2700036 #define TCP_FLAG_PSH 2
2700037 #define TCP_FLAG_RST 4
2700038 #define TCP_FLAG_SYN 8
2700039 #define TCP_FLAG_FIN 16
2700040 //-----
2700041 #define TCP_TRY_READ 1 // 2^0 Wake up reading
2700042 // TCP process.
2700043 #define TCP_TRY_WRITE 2 // 2^1 Wake up writing
2700044 // TCP process.
2700045 //-----
2700046 int tcp_tx_raw (h_port_t sport, h_port_t dport,
2700047                uint32_t seq, uint32_t ack_seq,
2700048                int flags,
2700049                h_addr_t saddr, h_addr_t daddr,
2700050                const void *buffer, size_t size);
2700051 int tcp_tx_sock (void *sock_item);
2700052 int tcp_tx_ack (void *sock_item);
2700053 int tcp_rx_ack (void *sock_item, void *packet);
2700054 int tcp (void);
2700055 int tcp_connect (void *sock_item);
2700056 int tcp_tx_rst (void *ip_packet);
2700057 void tcp_test (void);
2700058 void tcp_show (h_addr_t src, h_addr_t dst,
2700059               const struct tcphdr *tcphdr);
2700060 int tcp_close (void *sock_item);
2700061 int tcp_rx_data (void *sock_item, void *packet);
2700062 int tcp_status (void *ip_packet);
2700063 //-----
2700064 #endif

```

94.12.41 kernel/net/tcp/tcp.c



Si veda la sezione [93.23](#).

```

2710001 #include <stdlib.h>
2710002 #include <string.h>

```

```
2710003 #include <netinet/ip.h>
2710004 #include <netinet/tcp.h>
2710005 #include <kernel/net.h>
2710006 #include <kernel/net/tcp.h>
2710007 #include <kernel/fs.h>
2710008 #include <kernel/lib_s.h>
2710009 #include <kernel/lib_k.h>
2710010 #include <errno.h>
2710011 //-----
2710012 #define DEBUG 0
2710013 //-----
2710014 int
2710015 tcp (void)
2710016 {
2710017     int s;           // Socket table index.
2710018     int p;           // IP table index.
2710019     int q;           // Queue index.
2710020     int status;
2710021     struct tcphdr *tcp;
2710022     struct iphdr *ip;
2710023     int sfdn;        // New socket.
2710024     fd_t *sfd;
2710025     sock_t *sock;
2710026     struct sockaddr_in sa;
2710027     clock_t delay;
2710028     uint8_t *recv_data;
2710029     size_t recv_size;
2710030     int ret = 0;
2710031     unsigned int lseq;
2710032     //
2710033     // Scan local sockets.
2710034     //
2710035     for (s = 0; s < SOCK_MAX_SLOTS; s++)
2710036     {
2710037         if (!sock_table[s].active)
2710038             continue;
2710039         if (sock_table[s].family != AF_INET)
```

```
2710040     continue;
2710041     if (sock_table[s].protocol != IPPROTO_TCP)
2710042         continue;
2710043     if (sock_table[s].unreach_port)
2710044         continue;
2710045     if (sock_table[s].unreach_host)
2710046         continue;
2710047     //
2710048     // Calculate the delay from the last send.
2710049     //
2710050     delay = s_clock ((pid_t) 0) - sock_table[s].tcp.clock;
2710051     //
2710052     // Have we received something? Scan the
2710053     // ip_table[] to find a
2710054     // TCP packet that was not already seen by the
2710055     // socket.
2710056     //
2710057     for (p = 0; p < IP_MAX_PACKETS; p++)
2710058     {
2710059         // //////////////////////////////////////
2710060         // PACKET CHECK
2710061         // //////////////////////////////////////
2710062         //
2710063         // Check the protocol.
2710064         //
2710065         if (ip_table[p].packet.header.protocol !=
2710066             IPPROTO_TCP)
2710067             {
2710068                 //
2710069                 // It is not TCP.
2710070                 //
2710071                 continue;
2710072             }
2710073         //
2710074         // Is the packet new for the socket?
2710075         //
2710076         if (ip_table[p].clock <=
```

```
2710077         sock_table[s].read.clock[p])
2710078     {
2710079         //
2710080         // Already seen or packet too old.
2710081         //
2710082         continue;
2710083     }
2710084     //
2710085     // Get a pointer to IP and TCP headers.
2710086     //
2710087     ip = (struct iphdr *) &ip_table[p].packet.header;
2710088     tcp =
2710089         (struct tcphdr *) &ip_table[p].packet.
2710090         octet[ip->ihl * 4];
2710091     //
2710092     // Verify the ports.
2710093     //
2710094     if (tcp->dest != htons (sock_table[s].lport))
2710095     {
2710096         //
2710097         // The local port does not match!
2710098         //
2710099         continue;
2710100     }
2710101     //
2710102     if (tcp->source != htons (sock_table[s].rport))
2710103     {
2710104         //
2710105         // The remote port does not match, but
2710106         // might be
2710107         // listening and the packet might be a
2710108         // SYN.
2710109         //
2710110         if (sock_table[s].rport == 0
2710111             && sock_table[s].tcp.conn ==
2710112             TCP_LISTEN && tcp->syn && !tcp->ack)
2710113         {
```

```
2710114          //
2710115          // We hope that it is the first SYN.
2710116          //
2710117          ;
2710118      }
2710119      else
2710120      {
2710121          continue;
2710122      }
2710123  }
2710124  //
2710125  // Verify the IP addresses.
2710126  //
2710127  if (ip_table[p].packet.header.daddr
2710128      != htonl (sock_table[s].laddr))
2710129      {
2710130          //
2710131          // The local address does not match, but
2710132          // might be zero.
2710133          //
2710134          if (sock_table[s].laddr != 0)
2710135              {
2710136                  //
2710137                  // The local address is not zero, so
2710138                  // the match fails.
2710139                  //
2710140                  continue;
2710141              }
2710142      }
2710143  //
2710144  if (ip_table[p].packet.header.saddr
2710145      != htonl (sock_table[s].raddr))
2710146      {
2710147          //
2710148          // The remote address does not match,
2710149          // but the socket
2710150          // might be listening and che packet
```



```
2710151         // might be a SYN.
2710152         //
2710153         if (sock_table[s].raddr == 0
2710154             && sock_table[s].tcp.conn ==
2710155             TCP_LISTEN && tcp->syn && !tcp->ack)
2710156             {
2710157                 //
2710158                 // We hope that it is the first SYN.
2710159                 //
2710160                 ;
2710161             }
2710162         else
2710163             {
2710164                 continue;
2710165             }
2710166     }
2710167     //
2710168     // This TCP packet is new for the socket:
2710169     // save the clock time, so that the
2710170     // same packet is not read again.
2710171     //
2710172     sock_table[s].read.clock[p] = ip_table[p].clock;
2710173     //
2710174     // //////////////////////////////////////
2710175     // TCP PROTOCOL
2710176     // //////////////////////////////////////
2710177     //
2710178     if (DEBUG)
2710179         {
2710180             tcp_show (ntohl (ip->saddr),
2710181                     ntohl (ip->daddr), tcp);
2710182         }
2710183     //
2710184     recv_data = &((uint8_t *) tcp)[tcp->doff * 4];
2710185     recv_size =
2710186         ntohs (ip->tot_len) - (ip->ihl * 4) -
2710187         (tcp->doff * 4);
```

```
2710188 //
2710189 // Now we have received a TCP packet for the
2710190 // current
2710191 // socket, and we should do something with
2710192 // it...
2710193 //
2710194 if (tcp->rst)
2710195 {
2710196 //
2710197 // We have received a reset... What is
2710198 // resetting?
2710199 //
2710200 if ((sock_table[s].tcp.
2710201      rsq[sock_table[s].tcp.rsqi] ==
2710202      ntohs (tcp->seq)) || (tcp->ack
2710203                          &&
2710204                          (sock_table[s].tcp.
2710205                           lsq_ack ==
2710206                           ntohs (tcp->
2710207                               ack_seq))))
2710208 {
2710209     sock_table[s].tcp.recv_closed = 1;
2710210     sock_table[s].tcp.send_closed = 1;
2710211     sock_table[s].tcp.conn = TCP_RESET;
2710212     if (DEBUG)
2710213     {
2710214         k_printf ("%s] TCP_RESET\n",
2710215                 __func__);
2710216     }
2710217 }
2710218 else
2710219 {
2710220     k_printf ("lsq_ack=%3, rsq=%3\n",
2710221             sock_table[s].tcp.lsq_ack,
2710222             sock_table[s].tcp.
2710223             rsq[sock_table[s].tcp.rsqi]);
2710224 }
```

```
2710225     }
2710226     else if (sock_table[s].tcp.conn == 0
2710227             || sock_table[s].tcp.conn ==
2710228             TCP_CLOSE
2710229             || sock_table[s].tcp.conn == TCP_RESET)
2710230     {
2710231         //
2710232         // The connection is not yet ready or it
2710233         // is closed.
2710234         // We are not waiting any packet, so we
2710235         // just reject it.
2710236         //
2710237         tcp_tx_rst (ip);
2710238     }
2710239     else if (sock_table[s].tcp.conn == TCP_LISTEN)
2710240     {
2710241         //
2710242         // It should be a first SYN packet.
2710243         //
2710244         if (tcp->syn && !tcp->ack)
2710245         {
2710246             //
2710247             // Should be a new connection
2710248             // attempt. Can we queue
2710249             // it?
2710250             //
2710251             for (q = 0;
2710252                  q <
2710253                  sock_table[s].tcp.listen_max; q++)
2710254             {
2710255                 if (sock_table[s].tcp.
2710256                     listen_queue[q] == -1)
2710257                 {
2710258                     break;
2710259                 }
2710260             }
2710261             if (q >= sock_table[s].tcp.listen_max)
```

```
2710262         {
2710263             //
2710264             // The queue is full.
2710265             //
2710266             tcp_tx_rst (ip);
2710267             //
2710268             // Next packet.
2710269             //
2710270             continue;
2710271         }
2710272         //
2710273         // Is this connection attempt
2710274         // already done?
2710275         //
2710276         status = tcp_status (ip);
2710277         //
2710278         if (status < 0)
2710279             {
2710280                 //
2710281                 // Should not happen.
2710282                 //
2710283                 errset (errno);
2710284                 perror (NULL);
2710285                 //
2710286                 // Ignore the packet?!
2710287                 //
2710288                 continue;
2710289             }
2710290         else if (status == TCP_SYN_SENT)
2710291             {
2710292                 //
2710293                 // The same SYN was already
2710294                 // received and serviced:
2710295                 // just ignore the packet.
2710296                 //
2710297                 continue;
2710298             }
```

```
2710299     else if (status > 0)
2710300     {
2710301         //
2710302         // There is already a connection
2710303         // with the same
2710304         // addresses: ignore the SYN.
2710305         //
2710306         continue;
2710307     }
2710308     //
2710309     // The SYN is new!
2710310     // Can we open a new socket?
2710311     //
2710312     sfdn =
2710313         s_socket (sock_table[s].tcp.listen_pid,
2710314                 AF_INET, SOCK_STREAM,
2710315                 IPPROTO_TCP);
2710316     if (sfdn < 0)
2710317     {
2710318         //
2710319         // No, sorry.
2710320         //
2710321         tcp_tx_rst (ip);
2710322         //
2710323         // Next packet.
2710324         //
2710325         continue;
2710326     }
2710327     //
2710328     // Can we bind it to the same
2710329     // destination of the
2710330     // received packet?
2710331     //
2710332     sa.sin_family = AF_INET;
2710333     sa.sin_port = tcp->dest;
2710334     sa.sin_addr.s_addr = ip->daddr;
2710335     status =
```

```
2710336         s_bind (sock_table[s].tcp.listen_pid,
2710337                 sfdn, (struct sockaddr *) &sa,
2710338                 sizeof (sa));
2710339     if (status < 0)
2710340     {
2710341         //
2710342         // No, sorry.
2710343         //
2710344         tcp_tx_rst (ip);
2710345         close (sfdn);
2710346         //
2710347         // Next packet.
2710348         //
2710349         continue;
2710350     }
2710351     //
2710352     // Ok. Save the new socket number in
2710353     // queue.
2710354     //
2710355     sock_table[s].tcp.listen_queue[q] = sfdn;
2710356     //
2710357     // Prepare some pointers to reach
2710358     // the new socket
2710359     // easily.
2710360     //
2710361     sfd =
2710362         fd_reference (sock_table[s].tcp.
2710363                     listen_pid, &sfdn);
2710364     sock = sfd->file->sock;
2710365     //
2710366     // Connect the new socket with the
2710367     // remote node.
2710368     //
2710369     sock->raddr = ntohl (ip->saddr);
2710370     sock->rport = ntohs (tcp->source);
2710371     //
2710372     // The new socket has seen this SYN
```

```
2710373 // packet.
2710374 //
2710375 sock->read.clock[p] = ip_table[p].clock;
2710376 //
2710377 // Make the new socket answare with
2710378 // a second
2710379 // SYN.
2710380 //
2710381 memset (sock->tcp.lsqr, 0x00,
2710382         sizeof (sock->tcp.lsqr));
2710383 memset (sock->tcp.rsqr, 0x00,
2710384         sizeof (sock->tcp.rsqr));
2710385 srand ((unsigned int)
2710386        s_clock ((pid_t) 0));
2710387 lseq = rand ();
2710388 sock->tcp.lsqr_ack = lseq + 1;
2710389 sock->tcp.lsqr[++sock->tcp.lsqi] = lseq;
2710390 sock->tcp.rsqr[++sock->tcp.rsqi] =
2710391     ntohs (tcp->seq);
2710392 sock->tcp.rsqr[++sock->tcp.rsqi] =
2710393     ntohs (tcp->seq) + 1;
2710394 //
2710395 sock->tcp.can_send = 1;
2710396 sock->tcp.send_flags =
2710397     TCP_FLAG_SYN | TCP_FLAG_ACK;
2710398 if (DEBUG)
2710399     {
2710400         k_printf
2710401             ("%s] New conn. seq=%3u, "
2710402             "lsqr_ack=%3u\n",
2710403             __func__, lseq, sock->tcp.lsqr_ack);
2710404     }
2710405 tcp_tx_sock (sock);
2710406 //
2710407 // Put the new socket to
2710408 // TCP_SYN_RECV status.
2710409 //
```

```
2710410         sock->tcp.conn = TCP_SYN_RECV;
2710411     }
2710412     else
2710413     {
2710414         //
2710415         // We are listening: cannot accept
2710416         // other type of
2710417         // packets.
2710418         //
2710419         tcp_tx_rst (ip);
2710420     }
2710421 }
2710422 else if (sock_table[s].tcp.conn == TCP_SYN_SENT)
2710423 {
2710424     //
2710425     // It should be a second SYN packet with
2710426     // ACK.
2710427     //
2710428     if (tcp->syn
2710429         && tcp->ack
2710430         && tcp_rx_ack (&sock_table[s], ip) == 0)
2710431     {
2710432         //
2710433         // SYN + ACK.
2710434         //
2710435         // Save the initial remote sequence,
2710436         // because it
2710437         // is the first one, and save also
2710438         // the remote
2710439         // sequence that we will expect next
2710440         // time.
2710441         //
2710442         sock_table[s].tcp.rsq[++sock_table[s].
2710443                               tcp.rsqi] =
2710444             ntohl (tcp->seq);
2710445         sock_table[s].tcp.rsq[++sock_table[s].
2710446                               tcp.rsqi] =
```



```
2710447         ntohs (tcp->seq) + 1;
2710448         //
2710449         // The received SYN is to be
2710450         // confirmed with ACK.
2710451         // The expected next local sequence
2710452         // does not change,
2710453         // and in effect, we don't expect
2710454         // any other ACK back.
2710455         //
2710456         sock_table[s].tcp.lsq[++sock_table[s].
2710457                               tcp.lsqi] =
2710458             sock_table[s].tcp.lsq_ack;
2710459         tcp_tx_ack (&sock_table[s]);
2710460         //
2710461         // We are now in TCP_ESTABLISHED.
2710462         //
2710463         sock_table[s].tcp.conn = TCP_ESTABLISHED;
2710464         //
2710465         // Now the process can write and can
2710466         // receive
2710467         // data (can write --but cannot
2710468         // send-- and can
2710469         // receive --but cannot read--).
2710470         //
2710471         sock_table[s].tcp.can_write = 1;
2710472         sock_table[s].tcp.can_send = 0;
2710473         //
2710474         sock_table[s].tcp.can_recv = 1;
2710475         sock_table[s].tcp.can_read = 0;
2710476         //
2710477         ret |= TCP_TRY_WRITE;
2710478     }
2710479     //
2710480     // The case of a single ACK and a single
2710481     // SYN is not
2710482     // taken into consideration!
2710483     //
```

```
2710484         // No other type of packet is expected
2710485         // here.
2710486         //
2710487     }
2710488     else if (sock_table[s].tcp.conn == TCP_SYN_RECV)
2710489     {
2710490         //
2710491         // We are waiting an ACK for our second
2710492         // SYN.
2710493         //
2710494         if (tcp->ack
2710495             && tcp_rx_ack (&sock_table[s], ip) == 0)
2710496         {
2710497             //
2710498             // ACK ok.
2710499             // The connection is ready.
2710500             //
2710501             sock_table[s].tcp.conn = TCP_ESTABLISHED;
2710502             //
2710503             // Now the process can write and can
2710504             // receive
2710505             // data (can write --but cannot
2710506             // send-- and can
2710507             // receive --but cannot read--).
2710508             //
2710509             sock_table[s].tcp.can_write = 1;
2710510             sock_table[s].tcp.can_send = 0;
2710511             //
2710512             sock_table[s].tcp.can_recv = 1;
2710513             sock_table[s].tcp.can_read = 0;
2710514             //
2710515             ret |= TCP_TRY_WRITE;
2710516         }
2710517         //
2710518         // No other type of packet is expected
2710519         // here.
2710520         //
```

```
2710521     }
2710522     else if (sock_table[s].tcp.conn ==
2710523             TCP_ESTABLISHED)
2710524     {
2710525         //
2710526         // Might be a repeated SYN + ACK,
2710527         // because the other
2710528         // side don't have received our ACK.
2710529         // Just resend.
2710530         //
2710531         if (tcp->syn && tcp->ack)
2710532         {
2710533             tcp_tx_ack (&sock_table[s]);
2710534             //
2710535             // Next packet.
2710536             //
2710537             continue;
2710538         }
2710539         //
2710540         // It might be a normal ACK.
2710541         //
2710542         if (tcp->ack)
2710543         {
2710544             //
2710545             // Verify if the packet contains
2710546             // data: if there is
2710547             // data, before sending the ACK,
2710548             // must verify to be
2710549             // able to receive such data.
2710550             //
2710551             if (recv_size > 0)
2710552             {
2710553                 // k_printf ("[%i]", (int)
2710554                 // recv_size);
2710555                 //
2710556                 // There is data.
2710557                 //
```

```
2710558         if (!sock_table[s].tcp.can_recv)
2710559             {
2710560                 //
2710561                 // At the moment, cannot
2710562                 // receive: the packet
2710563                 // is currently ignored and
2710564                 // no ACK is sent.
2710565                 //
2710566                 continue;
2710567             }
2710568     }
2710569     //
2710570     // The received packet is empty or
2710571     // it can be received.
2710572     //
2710573     if (tcp_rx_ack (&sock_table[s], ip) == 0)
2710574         {
2710575             //
2710576             // ACK ok.
2710577             // The process can continue to
2710578             // write and the packet
2710579             // don't have to be resent.
2710580             //
2710581             sock_table[s].tcp.can_write = 1;
2710582             sock_table[s].tcp.can_send = 0;
2710583             //
2710584             ret |= TCP_TRY_WRITE;
2710585         }
2710586     else
2710587         {
2710588             //
2710589             // Next packet.
2710590             //
2710591             continue;
2710592         }
2710593     }
2710594     //
```

```
2710595 // It might be a FIN.
2710596 //
2710597 if (tcp->fin)
2710598 {
2710599 //
2710600 // Is the FIN in the right sequence?
2710601 //
2710602 if (sock_table[s].tcp.
2710603     rsq[sock_table[s].tcp.rsqi] ==
2710604     ntohl (tcp->seq))
2710605 {
2710606 //
2710607 // Yes, it is: close receiving.
2710608 //
2710609 sock_table[s].tcp.recv_closed = 1;
2710610 //
2710611 // ACK.
2710612 //
2710613 sock_table[s].tcp.
2710614     rsq[++sock_table[s].tcp.rsqi] =
2710615     ntohl (tcp->seq) + 1;
2710616 sock_table[s].tcp.
2710617     lsq[++sock_table[s].tcp.lsqi] =
2710618     sock_table[s].tcp.lsq_ack;
2710619 tcp_tx_ack (&sock_table[s]);
2710620 //
2710621 // Change status.
2710622 //
2710623 if (DEBUG)
2710624 {
2710625     k_printf
2710626         ("%s] TCP_CLOSE_WAIT\n",
2710627         __func__);
2710628 }
2710629 sock_table[s].tcp.conn =
2710630     TCP_CLOSE_WAIT;
2710631 }
```

```
2710632         else
2710633             {
2710634                 //
2710635                 // Just ignore it and jump to
2710636                 // the next packet.
2710637                 //
2710638                 continue;
2710639             }
2710640     }
2710641     //
2710642     // The received packet might contain
2710643     // some data, but can
2710644     // accept data only if the receiving
2710645     // buffer is empty
2710646     // (was already read from the reading
2710647     // process).
2710648     //
2710649     tcp_rx_data (&sock_table[s], ip);
2710650     //
2710651     ret |= TCP_TRY_READ;
2710652 }
2710653 else if (sock_table[s].tcp.conn == TCP_CLOSE_WAIT)
2710654 {
2710655     //
2710656     // It might be an ACK.
2710657     //
2710658     if (tcp->ack)
2710659     {
2710660         if (tcp_rx_ack (&sock_table[s], ip) == 0)
2710661         {
2710662             //
2710663             // ACK ok.
2710664             // The process can continue to
2710665             // write and the packet
2710666             // don't have to be resent.
2710667             //
2710668             sock_table[s].tcp.can_write = 1;
```

```
2710669         sock_table[s].tcp.can_send = 0;
2710670         //
2710671         ret |= TCP_TRY_WRITE;
2710672     }
2710673     else
2710674     {
2710675         //
2710676         // Next packet.
2710677         //
2710678         continue;
2710679     }
2710680 }
2710681 //
2710682 // The data coming from the outside is
2710683 // not taken anymore.
2710684 //
2710685 }
2710686 else if (sock_table[s].tcp.conn == TCP_LAST_ACK)
2710687 {
2710688     //
2710689     // It might be the final ACK.
2710690     //
2710691     if (tcp->ack)
2710692     {
2710693         if (tcp_rx_ack (&sock_table[s], ip) == 0)
2710694         {
2710695             //
2710696             // ACK ok. The two directions
2710697             // are closed and
2710698             // the packet confirmed don't
2710699             // have to be resent.
2710700             //
2710701             sock_table[s].tcp.recv_closed = 1;
2710702             sock_table[s].tcp.send_closed = 1;
2710703             sock_table[s].tcp.can_send = 0;
2710704             //
2710705             // Change status.
```

```
2710706 //
2710707 if (DEBUG)
2710708 {
2710709     k_printf ("%s] TCP_CLOSE\n",
2710710               __func__);
2710711 }
2710712 sock_table[s].tcp.conn = TCP_CLOSE;
2710713 }
2710714 else
2710715 {
2710716     //
2710717     // Next packet.
2710718     //
2710719     continue;
2710720 }
2710721 }
2710722 //
2710723 // The data coming from the outside is
2710724 // not taken anymore.
2710725 //
2710726 }
2710727 else if (sock_table[s].tcp.conn == TCP_FIN_WAIT1)
2710728 {
2710729     if (tcp->ack && tcp->fin)
2710730     {
2710731         //
2710732         // ACK and FIN
2710733         //
2710734         if (tcp_rx_ack (&sock_table[s], ip) == 0)
2710735         {
2710736             //
2710737             // ACK ok: the confirmed packet
2710738             // don't have to
2710739             // be resent.
2710740             //
2710741             sock_table[s].tcp.conn =
2710742                 TCP_FIN_WAIT2;
```



```
2710743         sock_table[s].tcp.can_send = 0;
2710744     }
2710745     else
2710746     {
2710747         //
2710748         // Next packet.
2710749         //
2710750         continue;
2710751     }
2710752     //
2710753     // Is the FIN in the right sequence?
2710754     //
2710755     if (sock_table[s].tcp.
2710756         rsq[sock_table[s].tcp.rsqi] ==
2710757         ntohl (tcp->seq))
2710758     {
2710759         //
2710760         // Yes, it is: close receiving.
2710761         //
2710762         sock_table[s].tcp.recv_closed = 1;
2710763         //
2710764         // ACK.
2710765         //
2710766         sock_table[s].tcp.
2710767             rsq[++sock_table[s].tcp.rsqi] =
2710768             ntohl (tcp->seq) + 1;
2710769         sock_table[s].tcp.
2710770             lsq[++sock_table[s].tcp.lsqi] =
2710771             sock_table[s].tcp.lsq_ack;
2710772         tcp_tx_ack (&sock_table[s]);
2710773         //
2710774         // Change status.
2710775         //
2710776         if (DEBUG)
2710777             {
2710778                 k_printf
2710779                     ("[%s] TCP_TIME_WAIT\n",
```

```
2710780         __func__);
2710781     }
2710782     sock_table[s].tcp.conn =
2710783         TCP_TIME_WAIT;
2710784     }
2710785     else
2710786     {
2710787         //
2710788         // Just ignore it and jump to
2710789         // the next packet.
2710790         //
2710791         continue;
2710792     }
2710793 }
2710794 else if (tcp->ack)
2710795 {
2710796     //
2710797     // ACK only.
2710798     //
2710799     if (tcp_rx_ack (&sock_table[s], ip) == 0)
2710800     {
2710801         //
2710802         // ACK ok: the confirmed packet
2710803         // don't have to
2710804         // be resent.
2710805         //
2710806         sock_table[s].tcp.conn =
2710807             TCP_FIN_WAIT2;
2710808         sock_table[s].tcp.can_send = 0;
2710809     }
2710810     else
2710811     {
2710812         //
2710813         // Next packet.
2710814         //
2710815         continue;
2710816     }
```

```
2710817     }
2710818     //
2710819     // The received packet might contain
2710820     // some data, but can
2710821     // accept data only if the receive
2710822     // channel is open and
2710823     // if the receiving buffer is empty.
2710824     //
2710825     tcp_rx_data (&sock_table[s], ip);
2710826     //
2710827     ret |= TCP_TRY_READ;
2710828 }
2710829 else if (sock_table[s].tcp.conn == TCP_FIN_WAIT2)
2710830 {
2710831     //
2710832     // It might be the final ACK.
2710833     //
2710834     if (tcp->fin && tcp->ack)
2710835     {
2710836         if (tcp_rx_ack (&sock_table[s], ip) == 0)
2710837         {
2710838             //
2710839             // ACK ok: the packet don't have
2710840             // to be resent.
2710841             //
2710842             sock_table[s].tcp.conn =
2710843                 TCP_TIME_WAIT;
2710844             sock_table[s].tcp.can_send = 0;
2710845             //
2710846             // Next packet.
2710847             //
2710848             continue;
2710849         }
2710850     else
2710851     {
2710852         //
2710853         // Next packet.
```

```
2710854         //
2710855         continue;
2710856     }
2710857 }
2710858 //
2710859 // The received packet might contain
2710860 // some data, but can
2710861 // accept data only if the receiving
2710862 // buffer is empty
2710863 // (was already read from the reading
2710864 // process).
2710865 //
2710866 tcp_rx_data (&sock_table[s], ip);
2710867 //
2710868 ret |= TCP_TRY_READ;
2710869 }
2710870 else if (sock_table[s].tcp.conn == TCP_TIME_WAIT)
2710871 {
2710872     //
2710873     // It might be duplicate final ACK.
2710874     //
2710875     if (tcp->fin && tcp->ack)
2710876     {
2710877         //
2710878         // Just resend the ACK.
2710879         //
2710880         tcp_tx_ack (&sock_table[s]);
2710881     }
2710882     //
2710883     // Close receiving too, if it is not
2710884     // already done.
2710885     //
2710886     sock_table[s].tcp.recv_closed = 1;
2710887     //
2710888     // Change status, without waiting
2710889     // anymore.
2710890     //
```

```

2710891         if (DEBUG)
2710892             {
2710893                 k_printf ("%s] TCP_CLOSE\n", __func__);
2710894             }
2710895         sock_table[s].tcp.conn = TCP_CLOSE;
2710896     }
2710897 }
2710898 //
2710899 // See if there is something to be re-sent (if
2710900 // the flag
2710901 // 'tcp.can_send' is set).
2710902 //
2710903 if (sock_table[s].tcp.can_send
2710904     && delay > TCP_MAX_DELAY)
2710905     {
2710906         tcp_tx_sock (&sock_table[s]);
2710907     }
2710908 }
2710909 //
2710910 // Return.
2710911 //
2710912 return (ret);
2710913 }

```

94.12.42 kernel/net/tcp/tcp_close.c

Si veda la sezione [93.23](#).

```

2720001 #include <stdlib.h>
2720002 #include <netinet/ip.h>
2720003 #include <netinet/tcp.h>
2720004 #include <errno.h>
2720005 #include <kernel/net.h>
2720006 #include <kernel/net/tcp.h>
2720007 #include <kernel/fs.h>
2720008 #include <kernel/lib_k.h>
2720009 //-----

```

```
2720010 #define DEBUG 0
2720011 //-----
2720012 int
2720013 tcp_close (void *sock_item)
2720014 {
2720015     sock_t *sock = sock_item;
2720016     //
2720017     // Is there already a connection?
2720018     //
2720019     if (sock->tcp.conn == 0 || sock->tcp.conn == TCP_CLOSE)
2720020     {
2720021         //
2720022         // Done or never opened.
2720023         //
2720024         return (0);
2720025     }
2720026     else if (sock->tcp.conn == TCP_RESET)
2720027     {
2720028         //
2720029         // Change to closed.
2720030         //
2720031         if (DEBUG)
2720032         {
2720033             k_printf ("%s] TCP_CLOSE\n", __func__);
2720034         }
2720035         sock->tcp.conn = TCP_CLOSE;
2720036         return (0);
2720037     }
2720038     else if (sock->tcp.conn == TCP_TIME_WAIT)
2720039     {
2720040         //
2720041         // Assume that the time is elapsed.
2720042         //
2720043         sock->tcp.conn = TCP_CLOSE;
2720044         if (DEBUG)
2720045         {
2720046             k_printf ("%s] TCP_CLOSE\n", __func__);
```

```
2720047     }
2720048     sock->tcp.conn = TCP_CLOSE;
2720049     return (0);
2720050 }
2720051 else if (sock->tcp.conn == TCP_FIN_WAIT1)
2720052 {
2720053     errset (EALREADY);
2720054     return (-1);
2720055 }
2720056 else if (sock->tcp.conn == TCP_FIN_WAIT2)
2720057 {
2720058     errset (EALREADY);
2720059     return (-1);
2720060 }
2720061 else if (sock->tcp.conn == TCP_ESTABLISHED)
2720062 {
2720063     sock->tcp.can_send = 1;
2720064     sock->tcp.lsq[++sock->tcp.lsqi] = sock->tcp.lsq_ack;
2720065     sock->tcp.send_flags = TCP_FLAG_FIN | TCP_FLAG_ACK;
2720066     tcp_tx_sock (sock);
2720067     sock->tcp.conn = TCP_FIN_WAIT1;
2720068     if (DEBUG)
2720069         k_printf ("%s] TCP_FIN_WAIT1\n", __func__);
2720070     errset (EINPROGRESS);
2720071     return (-1);
2720072 }
2720073 else if (sock->tcp.conn == TCP_CLOSE_WAIT)
2720074 {
2720075     sock->tcp.can_send = 1;
2720076     sock->tcp.lsq[++sock->tcp.lsqi] = sock->tcp.lsq_ack;
2720077     sock->tcp.send_flags = TCP_FLAG_FIN | TCP_FLAG_ACK;
2720078     tcp_tx_sock (sock);
2720079     sock->tcp.conn = TCP_LAST_ACK;
2720080     if (DEBUG)
2720081         k_printf ("%s] TCP_LAST_ACK\n", __func__);
2720082     errset (EINPROGRESS);
2720083     return (-1);
```

```
2720084     }
2720085     else
2720086     {
2720087         sock->tcp.can_send = 1;
2720088         sock->tcp.lsq[++sock->tcp.lsqi] = sock->tcp.lsq_ack;
2720089         sock->tcp.send_flags = TCP_FLAG_FIN | TCP_FLAG_ACK;
2720090         tcp_tx_sock (sock);
2720091         sock->tcp.conn = TCP_CLOSE;
2720092         if (DEBUG)
2720093             k_printf ("%s] TCP_CLOSE\n", __func__);
2720094         return (0);
2720095     }
2720096 }
```

94.12.43 kernel/net/tcp/tcp_connect.c

<<

Si veda la sezione [93.23](#).

```
2730001 #include <stdlib.h>
2730002 #include <netinet/ip.h>
2730003 #include <netinet/tcp.h>
2730004 #include <errno.h>
2730005 #include <kernel/net.h>
2730006 #include <kernel/net/tcp.h>
2730007 #include <kernel/fs.h>
2730008 #include <kernel/lib_k.h>
2730009 //-----
2730010 #define DEBUG 0
2730011 //-----
2730012 int
2730013 tcp_connect (void *sock_item)
2730014 {
2730015     sock_t *sock = sock_item;
2730016     unsigned int lseq;
2730017     //
2730018     // Is there already a connection?
2730019     //
```



```
2730020     if (sock->tcp.conn == 0 || sock->tcp.conn == TCP_CLOSE)
2730021     {
2730022         //
2730023         // There isn't.
2730024         //
2730025         memset (sock->tcp.lsq, 0x00, sizeof (sock->tcp.lsq));
2730026         memset (sock->tcp.rsq, 0x00, sizeof (sock->tcp.rsq));
2730027         srand ((unsigned int) s_clock ((pid_t) 0));
2730028         lseq = rand ();
2730029         sock->tcp.lsq_ack = lseq + 1;
2730030         sock->tcp.lsq[++sock->tcp.lsqi] = lseq;
2730031         //
2730032         sock->tcp.can_send = 1;
2730033         sock->tcp.send_size = 0;
2730034         sock->tcp.send_flags = TCP_FLAG_SYN;
2730035         tcp_tx_sock (sock);
2730036         //
2730037         sock->tcp.conn = TCP_SYN_SENT;
2730038         //
2730039         // The operation has begun and will take some
2730040         // time.
2730041         //
2730042         errset (EINPROGRESS);
2730043         return (-1);
2730044     }
2730045     else if (sock->tcp.conn == TCP_RESET)
2730046     {
2730047         //
2730048         // Cannot connect: the socket status is reset to
2730049         // closed, to let
2730050         // the process retry, if it is really willing to
2730051         // do it.
2730052         //
2730053         sock->tcp.conn = TCP_CLOSE;
2730054         errset (ECONNREFUSED);
2730055         return (-1);
2730056     }
```

```
2730057     else if (sock->tcp.conn == TCP_SYN_SENT
2730058             || sock->tcp.conn == TCP_SYN_RECV)
2730059     {
2730060         //
2730061         // Already in progress.
2730062         //
2730063         errset (EALREADY);
2730064         return (-1);
2730065     }
2730066     else if (sock->tcp.conn == TCP_ESTABLISHED)
2730067     {
2730068         //
2730069         // Connection established.
2730070         //
2730071         return (0);
2730072     }
2730073     else
2730074     {
2730075         //
2730076         // Cannot reconnect.
2730077         //
2730078         errset (EISCONN);
2730079         return (-1);
2730080     }
2730081 }
```

94.12.44 kernel/net/tcp/tcp_rx_ack.c

«

Si veda la sezione [93.23](#).

```
2740001 #include <kernel/net.h>
2740002 #include <kernel/net/ip.h>
2740003 #include <kernel/net/route.h>
2740004 #include <kernel/net/tcp.h>
2740005 #include <sys/os32.h>
2740006 #include <kernel/lib_k.h>
2740007 #include <kernel/fs.h>
```

```
2740008 #include <errno.h>
2740009 #include <arpa/inet.h>
2740010 #include <netinet/in.h>
2740011 #include <netinet/tcp.h>
2740012 //-----
2740013 #define DEBUG 0
2740014 //-----
2740015 int
2740016 tcp_rx_ack (void *sock_item, void *packet)
2740017 {
2740018     sock_t *sock = sock_item;
2740019     struct iphdr *iphdr = packet;
2740020     struct tcphdr *tcphdr = (struct tcphdr *)
2740021         &((uint8_t *) packet)[iphdr->ihl * 4];
2740022     int i;
2740023     //
2740024     // Is the ACK sequence right?
2740025     //
2740026     if (sock->tcp.lsq_ack != ntohl (tcphdr->ack_seq))
2740027     {
2740028         //
2740029         // If it is a previous sequence, just ignore
2740030         // the packet.
2740031         //
2740032         for (i = 0; i < 16; i++)
2740033         {
2740034             if (sock->tcp.lsq[i] == ntohl (tcphdr->ack_seq))
2740035             {
2740036                 break;
2740037             }
2740038         }
2740039         if (i >= 16)
2740040         {
2740041             if (DEBUG)
2740042             {
2740043                 k_printf ("ERR loc seq: ");
2740044                 tcp_show (ntohl (iphdr->saddr),
```

```
2740045         ntohs (iphdr->daddr), tcphdr);
2740046
2740047         int j;
2740048         for (j = 0; j < 16; j++)
2740049             {
2740050                 if (sock->tcp.lsq[j] != 0)
2740051                     {
2740052                         k_printf ("%3u ", sock->tcp.lsq[j]);
2740053                     }
2740054             }
2740055         k_printf ("lsq_ack=%3u ", sock->tcp.lsq_ack);
2740056         k_printf ("\n");
2740057     }
2740058     //
2740059     // The ACK is out of sequence: sorry.
2740060     //
2740061     tcp_tx_rst (iphdr);
2740062 }
2740063 return (-1);
2740064 }
2740065 //
2740066 // Is the TCP sequence what we expected? But notice
2740067 // that, if our
2740068 // remote expected sequence is zero, this is the
2740069 // first time that
2740070 // get it, so it is right.
2740071 //
2740072 if (sock->tcp.rsq[sock->tcp.rsqi] != 0
2740073     && sock->tcp.rsq[sock->tcp.rsqi] !=
2740074     ntohs (tcphdr->seq))
2740075     {
2740076         //
2740077         // If it is a previous sequence, just ignore
2740078         // the packet.
2740079         //
2740080         for (i = 0; i < 16; i++)
2740081             {
```

```
2740082         if (sock->tcp.rsq[i] == ntohl (tcphdr->seq))
2740083             {
2740084                 break;
2740085             }
2740086     }
2740087     if (i >= 16)
2740088     {
2740089         //
2740090         // The packet is out of sequence: sorry.
2740091         //
2740092         if (DEBUG)
2740093         {
2740094             k_printf ("ERR rem seq: ");
2740095             tcp_show (ntohl (iphdr->saddr),
2740096                     ntohl (iphdr->daddr), tcphdr);
2740097
2740098             int j;
2740099             for (j = 0; j < 16; j++)
2740100                 {
2740101                     if (sock->tcp.rsq[j] != 0)
2740102                         {
2740103                             k_printf ("%3u ", sock->tcp.rsq[j]);
2740104                         }
2740105                 }
2740106             k_printf ("\n");
2740107         }
2740108         tcp_tx_rst (iphdr);
2740109     }
2740110     return (-1);
2740111 }
2740112 return (0);
2740113 }
```

94.12.45 kernel/net/tcp/tcp_rx_data.c



Si veda la sezione [93.23](#).

```
2750001 #include <kernel/net.h>
2750002 #include <kernel/net/ip.h>
2750003 #include <kernel/net/route.h>
2750004 #include <kernel/net/tcp.h>
2750005 #include <sys/os32.h>
2750006 #include <kernel/lib_k.h>
2750007 #include <kernel/fs.h>
2750008 #include <errno.h>
2750009 #include <arpa/inet.h>
2750010 #include <netinet/in.h>
2750011 #include <netinet/tcp.h>
2750012 //-----
2750013 #define DEBUG 0
2750014 //-----
2750015 int
2750016 tcp_rx_data (void *sock_item, void *packet)
2750017 {
2750018     sock_t *sock = sock_item;
2750019     struct iphdr *iphdr = packet;
2750020     struct tcphdr *tcphdr = (struct tcphdr *)
2750021         &((uint8_t *) packet)[iphdr->ihl * 4];
2750022     uint8_t *recv_data;
2750023     size_t recv_size;
2750024     //
2750025     recv_data = &((uint8_t *) tcphdr)[tcphdr->doff * 4];
2750026     recv_size = ntohs (iphdr->tot_len) - (iphdr->ihl * 4)
2750027         - (tcphdr->doff * 4);
2750028     //
2750029     if (DEBUG)
2750030     {
2750031         if (recv_size > 0 && !sock->tcp.can_recv)
2750032         {
2750033             k_printf ("%s] not ready to get data\n",
2750034                 __func__);
```

```
2750035     }
2750036 }
2750037 //
2750038 // If we receive zero data, it is ok.
2750039 //
2750040 if (recv_size == 0)
2750041 {
2750042     //
2750043     // Nothing to do, but there is no error.
2750044     //
2750045     return (0);
2750046 }
2750047 if (recv_size < 0)
2750048 {
2750049     return (-1);
2750050 }
2750051 if (recv_size > 0 && !sock->tcp.can_recv)
2750052 {
2750053     return (-1);
2750054 }
2750055 //
2750056 // Check the size.
2750057 //
2750058 if (recv_size > sizeof (sock->tcp.recv_data))
2750059 {
2750060     k_printf ("%s] cannot accept a packet "
2750061              "payload of %u bytes; packet "
2750062              "truncated at %u bytes!\n",
2750063              __func__, recv_size,
2750064              sizeof (sock->tcp.recv_data));
2750065     //
2750066     recv_size = sizeof (sock->tcp.recv_data);
2750067 }
2750068 //
2750069 memcpy (sock->tcp.recv_data, recv_data, recv_size);
2750070 sock->tcp.recv_size = recv_size;
2750071 sock->tcp.recv_index = sock->tcp.recv_data;
```

```
2750072 //
2750073 // Must ACK back for the data received.
2750074 //
2750075 // The remote sequence that we will expect next time
2750076 // and the local sequence, expected from the other
2750077 // side.
2750078 //
2750079 sock->tcp.rsq[++sock->tcp.rsqi] =
2750080     ntohs (tcphdr->seq) + recv_size;
2750081 sock->tcp.lsq[++sock->tcp.lsqi] = sock->tcp.lsq_ack;
2750082 //
2750083 tcp_tx_ack (sock);
2750084 //
2750085 // Now the received data, if any, is to be read.
2750086 //
2750087 sock->tcp.can_recv = 0;
2750088 sock->tcp.can_read = 1;
2750089 //
2750090 //
2750091 //
2750092 return (0);
2750093 }
```

94.12.46 kernel/net/tcp/tcp_show.c

«

Si veda la sezione [93.23](#).

```
2760001 #include <kernel/net.h>
2760002 #include <kernel/net/ip.h>
2760003 #include <kernel/net/route.h>
2760004 #include <kernel/net/tcp.h>
2760005 #include <sys/os32.h>
2760006 #include <kernel/lib_k.h>
2760007 #include <kernel/fs.h>
2760008 #include <errno.h>
2760009 #include <arpa/inet.h>
2760010 #include <netinet/in.h>
```



```
2760011 #include <netinet/tcp.h>
2760012 //-----
2760013 void
2760014 tcp_show (h_addr_t src, h_addr_t dst,
2760015           const struct tcphdr *tcphdr)
2760016 {
2760017     struct in_addr addr_1;
2760018     struct in_addr addr_2;
2760019     char addr_string_1[INET_ADDRSTRLEN];
2760020     char addr_string_2[INET_ADDRSTRLEN];
2760021     //
2760022     if (tcphdr == NULL)
2760023     {
2760024         return;
2760025     }
2760026     //
2760027     addr_1.s_addr = htonl (src);
2760028     addr_2.s_addr = htonl (dst);
2760029     inet_ntop (AF_INET, &addr_1, addr_string_1,
2760030               (socklen_t) sizeof (addr_string_1));
2760031     inet_ntop (AF_INET, &addr_2, addr_string_2,
2760032               (socklen_t) sizeof (addr_string_2));
2760033     k_printf ("TCP %s:%i > %s:%i ", addr_string_1,
2760034              (unsigned int) ntohs (tcphdr->source),
2760035              addr_string_2,
2760036              (unsigned int) ntohs (tcphdr->dest));
2760037     k_printf ("s=%3u ", ntohl (tcphdr->seq));
2760038     k_printf ("k=%3u ", ntohl (tcphdr->ack_seq));
2760039
2760040     if (tcphdr->ack)
2760041         k_printf ("ack ");
2760042     if (tcphdr->psh)
2760043         k_printf ("psh ");
2760044     if (tcphdr->rst)
2760045         k_printf ("rst ");
2760046     if (tcphdr->syn)
2760047         k_printf ("syn ");
```

```
2760048     if (tcphdr->fin)
2760049         k_printf ("fin ");
2760050
2760051     k_printf ("\n");
2760052 }
```

94.12.47 kernel/net/tcp/tcp_status.c

<<

Si veda la sezione [93.23](#).

```
2770001 #include <kernel/net.h>
2770002 #include <kernel/net/ip.h>
2770003 #include <kernel/net/route.h>
2770004 #include <kernel/net/tcp.h>
2770005 #include <sys/os32.h>
2770006 #include <kernel/lib_k.h>
2770007 #include <kernel/fs.h>
2770008 #include <errno.h>
2770009 #include <stdlib.h>
2770010 #include <arpa/inet.h>
2770011 #include <netinet/in.h>
2770012 #include <netinet/tcp.h>
2770013 //-----
2770014 int
2770015 tcp_status (void *ip_packet)
2770016 {
2770017     struct iphdr *iphdr;
2770018     struct tcphdr *tcphdr;
2770019     int s;
2770020 //
2770021     if (ip_packet == NULL)
2770022     {
2770023         errset (EINVAL);
2770024         return (-1);
2770025     }
2770026 //
2770027     iphdr = ip_packet;
```

```
2770028     tcphdr = (struct tcphdr *)
2770029         &(((uint8_t *) ip_packet)[iphdr->ihl * 4]);
2770030     //
2770031     if (iphdr->saddr == 0 || iphdr->daddr == 0)
2770032     {
2770033         errset (EINVAL);
2770034         return (-1);
2770035     }
2770036     //
2770037     if (tcphdr->source == 0 || tcphdr->dest == 0)
2770038     {
2770039         errset (EINVAL);
2770040         return (-1);
2770041     }
2770042     //
2770043     // Find a connection with the same IPs and ports.
2770044     //
2770045     for (s = 0; s < SOCK_MAX_SLOTS; s++)
2770046     {
2770047         if (!sock_table[s].active)
2770048             continue;
2770049         if (sock_table[s].family != AF_INET)
2770050             continue;
2770051         if (sock_table[s].protocol != IPPROTO_TCP)
2770052             continue;
2770053         if (sock_table[s].unreach_port)
2770054             continue;
2770055         if (sock_table[s].unreach_host)
2770056             continue;
2770057         if (sock_table[s].laddr != ntohl (iphdr->daddr))
2770058             continue;
2770059         if (sock_table[s].lport != ntohs (tcphdr->dest))
2770060             continue;
2770061         if (sock_table[s].raddr != ntohl (iphdr->saddr))
2770062             continue;
2770063         if (sock_table[s].rport != ntohs (tcphdr->source))
2770064             continue;
```

```
2770065 //
2770066 // A corresponding socket was found.
2770067 //
2770068     return ((int) sock_table[s].tcp.conn);
2770069 }
2770070 //
2770071 // Socket not found.
2770072 //
2770073     return (0);
2770074 }
```

94.12.48 kernel/net/tcp/tcp_test.c



Si veda la sezione [93.23](#).

```
2780001 #include <kernel/driver/pci.h>
2780002 #include <kernel/net/ip.h>
2780003 #include <kernel/net/tcp.h>
2780004 #include <kernel/net.h>
2780005 #include <kernel/ibm_i386.h>
2780006 #include <errno.h>
2780007 #include <kernel/lib_k.h>
2780008 #include <kernel/lib_s.h>
2780009 #include <stdint.h>
2780010 //-----
2780011 void
2780012 tcp_test (void)
2780013 {
2780014
2780015     h_addr_t src = 0xAC150B10; // 172, 21, 11, 16
2780016     h_addr_t dst = 0xAC150B0F; // 172, 21, 11, 15
2780017
2780018     int status;
2780019 //
2780020 //
2780021 //
2780022     status = tcp_tx_raw (12345, 1234, 100000, 0,
```

```
2780023         TCP_FLAG_SYN,
2780024         src, dst, "SYN", (size_t) 4);
2780025
2780026     if (status)
2780027     {
2780028         k_perror (NULL);
2780029     }
2780030 }
```

94.12.49 kernel/net/tcp/tcp_tx_ack.c



Si veda la sezione [93.23](#).

```
2790001 #include <kernel/net.h>
2790002 #include <kernel/net/ip.h>
2790003 #include <kernel/net/route.h>
2790004 #include <kernel/net/tcp.h>
2790005 #include <sys/os32.h>
2790006 #include <kernel/lib_k.h>
2790007 #include <kernel/fs.h>
2790008 #include <errno.h>
2790009 #include <arpa/inet.h>
2790010 #include <netinet/in.h>
2790011 #include <netinet/tcp.h>
2790012 //-----
2790013 #define DEBUG 0
2790014 //-----
2790015 int
2790016 tcp_tx_ack (void *sock_item)
2790017 {
2790018     sock_t *sock = sock_item;
2790019     tcp_packet_t packet;
2790020     tcp_pseudo_header_t pseudo;
2790021     uint16_t checksum;
2790022 //
2790023     if (sock == NULL)
2790024     {
```

```
2790025     errset (EINVAL);
2790026     return (-1);
2790027 }
2790028 if (sock->laddr == 0 || sock->raddr == 0)
2790029 {
2790030     errset (EINVAL);
2790031     return (-1);
2790032 }
2790033 if (sock->lport == 0 || sock->rport == 0)
2790034 {
2790035     errset (EINVAL);
2790036     return (-1);
2790037 }
2790038 //
2790039 // Prepare the TCP packet.
2790040 //
2790041 memset (&packet.header, 0, sizeof (struct tcphdr));
2790042 //
2790043 packet.header.source = htons (sock->lport);
2790044 packet.header.dest = htons (sock->rport);
2790045 packet.header.seq = htonl (sock->tcp.lsqr[sock->tcp.lsqi]);
2790046 packet.header.ack_seq =
2790047     htonl (sock->tcp.rsqr[sock->tcp.rsqi]);
2790048 packet.header.doff = (sizeof (struct tcphdr) / 4);
2790049 packet.header.ack = 1;
2790050 packet.header.window = htons (TCP_MSS);           // Minimal
2790051 // window
2790052 packet.header.check = 0;
2790053 //
2790054 // Prepare the pseudo header.
2790055 //
2790056 pseudo.saddr = htonl (sock->laddr);
2790057 pseudo.daddr = htonl (sock->raddr);
2790058 pseudo.zero = 0;
2790059 pseudo.protocol = IPPROTO_TCP;
2790060 pseudo.length = htons (sizeof (struct tcphdr));
2790061 //
```

```
2790062 // Now set the header checksum.
2790063 //
2790064 checksum =
2790065     ~(ip_checksum
2790066         ((void *) &packet, sizeof (struct tcphdr),
2790067         (void *) &pseudo, sizeof (tcp_pseudo_header_t)));
2790068 if (checksum == 0)
2790069     {
2790070         checksum = 0xFFFF;
2790071     }
2790072 packet.header.check = htons (checksum);
2790073 //
2790074 // Send to the lower network level.
2790075 //
2790076 if (DEBUG)
2790077     {
2790078         tcp_show (sock->laddr, sock->raddr,
2790079                 (struct tcphdr *) &packet);
2790080     }
2790081 return (ip_tx
2790082         (sock->laddr, sock->raddr, (int) IPPROTO_TCP,
2790083         &packet, sizeof (struct tcphdr)));
2790084 }
```

94.12.50 kernel/net/tcp/tcp_tx_raw.c

Si veda la sezione [93.23](#).

```
2800001 #include <kernel/net.h>
2800002 #include <kernel/net/ip.h>
2800003 #include <kernel/net/route.h>
2800004 #include <kernel/net/tcp.h>
2800005 #include <sys/os32.h>
2800006 #include <kernel/lib_k.h>
2800007 #include <errno.h>
2800008 #include <arpa/inet.h>
2800009 #include <netinet/tcp.h>
```

```
280010 //-----
280011 #define DEBUG 0
280012 //-----
280013 int
280014 tcp_tx_raw (h_port_t sport, h_port_t dport,
280015             uint32_t seq, uint32_t ack_seq, int flags,
280016             h_addr_t saddr, h_addr_t daddr,
280017             const void *buffer, size_t size)
280018 {
280019     tcp_packet_t packet;
280020     tcp_pseudo_header_t pseudo;
280021     uint16_t checksum;
280022     size_t tcp_size = size + sizeof (struct tcphdr);
280023     //
280024     // Verify to have the source address: it is
280025     // necessary here for
280026     // the checksum calculation.
280027     //
280028     if (saddr == 0)
280029     {
280030         //
280031         // Default source address: get the source
280032         // address from the routing
280033         // table, based on the destination.
280034         //
280035         saddr = route_remote_to_local (daddr);
280036         if (saddr == ((h_addr_t) - 1))
280037         {
280038             errset (errno);
280039             return (-1);
280040         }
280041     }
280042     //
280043     // Prepare the TCP packet.
280044     //
280045     memset (&packet.header, 0, sizeof (struct tcphdr));
280046     //
```



```
2800047 packet.header.source = htons (sport);
2800048 packet.header.dest = htons (dport);
2800049 packet.header.seq = htonl (seq);
2800050 packet.header.ack_seq = htonl (ack_seq);
2800051 packet.header.doff = (sizeof (struct tcphdr) / 4);
2800052 if (flags & TCP_FLAG_ACK)
2800053     packet.header.ack = 1;
2800054 if (flags & TCP_FLAG_PSH)
2800055     packet.header.psh = 1;
2800056 if (flags & TCP_FLAG_RST)
2800057     packet.header.rst = 1;
2800058 if (flags & TCP_FLAG_SYN)
2800059     packet.header.syn = 1;
2800060 if (flags & TCP_FLAG_FIN)
2800061     packet.header.fin = 1;
2800062 if (flags & TCP_FLAG_RST)
2800063     {
2800064         packet.header.window = htons (0);
2800065     }
2800066 else
2800067     {
2800068         //
2800069         // Minimal window.
2800070         //
2800071         packet.header.window = htons (TCP_MSS);
2800072     }
2800073 packet.header.check = 0;
2800074 //
2800075 memcpy (packet.data, buffer, size);
2800076 //
2800077 // Prepare the pseudo header.
2800078 //
2800079 pseudo.saddr = htonl (saddr);
2800080 pseudo.daddr = htonl (daddr);
2800081 pseudo.zero = 0;
2800082 pseudo.protocol = IPPROTO_TCP;
2800083 pseudo.length = htons (tcp_size);
```

```

2800084 //
2800085 // Now set the header checksum.
2800086 //
2800087 checksum = ~(ip_checksum ((void *) &packet, tcp_size,
2800088                          (void *) &pseudo,
2800089                          sizeof (tcp_pseudo_header_t)));
2800090
2800091 if (checksum == 0)
2800092     {
2800093         checksum = 0xFFFF;
2800094     }
2800095 packet.header.check = htons (checksum);
2800096 //
2800097 // Send to the lower network level.
2800098 //
2800099 return (ip_tx
2800100         (saddr, daddr, (int) IPPROTO_TCP, &packet,
2800101         tcp_size));

```

94.12.51 kernel/net/tcp/tcp_tx_rst.c



Si veda la sezione [93.23](#).

```

2810001 #include <kernel/net.h>
2810002 #include <kernel/net/ip.h>
2810003 #include <kernel/net/route.h>
2810004 #include <kernel/net/tcp.h>
2810005 #include <sys/os32.h>
2810006 #include <kernel/lib_k.h>
2810007 #include <kernel/fs.h>
2810008 #include <errno.h>
2810009 #include <stdlib.h>
2810010 #include <arpa/inet.h>
2810011 #include <netinet/in.h>
2810012 #include <netinet/tcp.h>
2810013 //-----
2810014 #define DEBUG 0

```

```
2810015 //-----
2810016 int
2810017 tcp_tx_rst (void *ip_packet)
2810018 {
2810019     struct iphdr *iphdr = ip_packet;
2810020     struct tcphdr *tcphdr;
2810021     uint32_t seq;
2810022     uint32_t ack_seq;
2810023     tcp_packet_t packet;
2810024     tcp_pseudo_header_t pseudo;
2810025     uint16_t checksum;
2810026     //
2810027     if (ip_packet == NULL)
2810028     {
2810029         errset (EINVAL);
2810030         return (-1);
2810031     }
2810032     //
2810033     iphdr = ip_packet;
2810034     tcphdr = (struct tcphdr *)
2810035         &(((uint8_t *) ip_packet)[iphdr->ihl * 4]);
2810036     //
2810037     if (iphdr->saddr == 0 || iphdr->daddr == 0)
2810038     {
2810039         errset (EINVAL);
2810040         return (-1);
2810041     }
2810042     //
2810043     if (tcphdr->source == 0 || tcphdr->dest == 0)
2810044     {
2810045         errset (EINVAL);
2810046         return (-1);
2810047     }
2810048     //
2810049     // If the bad TCP packet has a ACK sequence, we
2810050     // replay with
2810051     // the same sequence (the one that the other side
```

```
2810052 // expects).
2810053 //
2810054 if (tcphdr->ack)
2810055 {
2810056     seq = ntohl (tcphdr->ack_seq);
2810057 }
2810058 else
2810059 {
2810060     seq = rand ();
2810061 }
2810062 //
2810063 // Our reset ACK has the same sequence received.
2810064 //
2810065 ack_seq = ntohl (tcphdr->seq);
2810066 //
2810067 // Prepare the TCP packet.
2810068 //
2810069 memset (&packet.header, 0, sizeof (struct tcphdr));
2810070 //
2810071 packet.header.source = tcphdr->dest;
2810072 packet.header.dest = tcphdr->source;
2810073 packet.header.seq = htonl (seq);
2810074 packet.header.ack_seq = htonl (ack_seq);
2810075 packet.header.doff = (sizeof (struct tcphdr) / 4);
2810076 packet.header.ack = 1;
2810077 packet.header.rst = 1;
2810078 packet.header.window = 0;
2810079 packet.header.check = 0;
2810080 //
2810081 // Prepare the pseudo header.
2810082 //
2810083 pseudo.saddr = iphdr->saddr;
2810084 pseudo.daddr = iphdr->daddr;
2810085 pseudo.zero = 0;
2810086 pseudo.protocol = IPPROTO_TCP;
2810087 pseudo.length = htons (sizeof (struct tcphdr));
2810088 //
```

```
2810089 // Now set the header checksum.
2810090 //
2810091 checksum =
2810092     ~(ip_checksum
2810093        ((void *) &packet, sizeof (struct tcphdr),
2810094         (void *) &pseudo, sizeof (tcp_pseudo_header_t)));
2810095 if (checksum == 0)
2810096     {
2810097         checksum = 0xFFFF;
2810098     }
2810099 packet.header.check = htons (checksum);
2810100 //
2810101 // Send to the lower network level.
2810102 //
2810103 if (DEBUG)
2810104     {
2810105         tcp_show (ntohl (iphdr->saddr),
2810106                  ntohl (iphdr->daddr),
2810107                  (struct tcphdr *) &packet);
2810108     }
2810109 return (ip_tx
2810110        (ntohl (iphdr->saddr), ntohl (iphdr->daddr),
2810111         IPPROTO_TCP, &packet, sizeof (struct tcphdr)));
2810112 }
```

94.12.52 kernel/net/tcp/tcp_tx_sock.c

Si veda la sezione [93.23](#).

```
2820001 #include <kernel/net.h>
2820002 #include <kernel/net/ip.h>
2820003 #include <kernel/net/route.h>
2820004 #include <kernel/net/tcp.h>
2820005 #include <sys/os32.h>
2820006 #include <kernel/lib_k.h>
2820007 #include <kernel/fs.h>
2820008 #include <errno.h>
```



```
282009 #include <arpa/inet.h>
282010 #include <netinet/in.h>
282011 #include <netinet/tcp.h>
282012 //-----
282013 #define DEBUG 0
282014 //-----
282015 int
282016 tcp_tx_sock (void *sock_item)
282017 {
282018     sock_t *sock = sock_item;
282019     tcp_packet_t packet;
282020     tcp_pseudo_header_t pseudo;
282021     uint16_t checksum;
282022     size_t tcp_size;
282023     uint32_t seq;
282024     uint32_t ack_seq;
282025     size_t send_size;
282026     //
282027     if (sock == NULL)
282028     {
282029         errset (EINVAL);
282030         return (-1);
282031     }
282032     if (!sock->tcp.can_send)
282033     {
282034         errset (EINVAL);
282035         return (-1);
282036     }
282037     if (sock->laddr == 0 || sock->raddr == 0)
282038     {
282039         errset (EINVAL);
282040         return (-1);
282041     }
282042     if (sock->lport == 0 || sock->rport == 0)
282043     {
282044         errset (EINVAL);
282045         return (-1);

```

```
2820046     }
2820047     //
2820048     // Sequences and size.
2820049     //
2820050     seq = sock->tcp.lsq[sock->tcp.lsqi];
2820051     if ((sock->tcp.send_flags & TCP_FLAG_SYN)
2820052         || (sock->tcp.send_flags & TCP_FLAG_FIN))
2820053     {
2820054         //
2820055         // A SYN or FIN packet cannot load data.
2820056         //
2820057         send_size = 0;
2820058         //
2820059         // The next expected ACK from the other side is
2820060         // just
2820061         // +1.
2820062         //
2820063         sock->tcp.lsq_ack = seq + 1;
2820064     }
2820065     else
2820066     {
2820067         send_size = sock->tcp.send_size;
2820068         //
2820069         // The next expected ACK from the other side is
2820070         // + size of the sent data.
2820071         //
2820072         sock->tcp.lsq_ack = seq + send_size;
2820073     }
2820074     //
2820075     if (sock->tcp.send_flags & TCP_FLAG_ACK)
2820076     {
2820077         ack_seq = sock->tcp.rsq[sock->tcp.rsqi];
2820078     }
2820079     else
2820080     {
2820081         ack_seq = 0;
2820082     }
```

```
2820083 //
2820084 // Prepare the TCP packet.
2820085 //
2820086 memset (&packet.header, 0, sizeof (struct tcphdr));
2820087 //
2820088 packet.header.source = htons (sock->lport);
2820089 packet.header.dest = htons (sock->rport);
2820090 packet.header.seq = htonl (seq);
2820091 packet.header.ack_seq = htonl (ack_seq);
2820092 packet.header.doff = (sizeof (struct tcphdr) / 4);
2820093 if (sock->tcp.send_flags & TCP_FLAG_ACK)
2820094     packet.header.ack = 1;
2820095 if (sock->tcp.send_flags & TCP_FLAG_PSH)
2820096     packet.header.psh = 1;
2820097 if (sock->tcp.send_flags & TCP_FLAG_RST)
2820098     packet.header.rst = 1;
2820099 if (sock->tcp.send_flags & TCP_FLAG_SYN)
2820100     packet.header.syn = 1;
2820101 if (sock->tcp.send_flags & TCP_FLAG_FIN)
2820102     packet.header.fin = 1;
2820103 if (sock->tcp.send_flags & TCP_FLAG_RST)
2820104     {
2820105         packet.header.window = htons (0);
2820106     }
2820107 else
2820108     {
2820109         //
2820110         // Minimal window.
2820111         //
2820112         packet.header.window = htons (TCP_MSS);
2820113     }
2820114 packet.header.check = 0;
2820115 //
2820116 memcpy (packet.data, sock->tcp.send_data, send_size);
2820117 //
2820118 tcp_size = sizeof (struct tcphdr) + send_size;
2820119 //
```



```
2820120 // Prepare the pseudo header.
2820121 //
2820122 pseudo.saddr = htonl (sock->laddr);
2820123 pseudo.daddr = htonl (sock->raddr);
2820124 pseudo.zero = 0;
2820125 pseudo.protocol = IPPROTO_TCP;
2820126 pseudo.length = htons (tcp_size);
2820127 //
2820128 // Now set the header checksum.
2820129 //
2820130 checksum = ~(ip_checksum ((void *) &packet, tcp_size,
2820131                          (void *) &pseudo,
2820132                          sizeof (tcp_pseudo_header_t)));
2820133 if (checksum == 0)
2820134     {
2820135         checksum = 0xFFFF;
2820136     }
2820137 packet.header.check = htons (checksum);
2820138 //
2820139 // Send to the lower network level.
2820140 //
2820141 if (DEBUG)
2820142     {
2820143         tcp_show (sock->laddr, sock->raddr,
2820144                 (struct tcphdr *) &packet);
2820145     }
2820146 sock->tcp.clock = s_clock ((pid_t) 0);
2820147 return (ip_tx
2820148         (sock->laddr, sock->raddr, (int) IPPROTO_TCP,
2820149         &packet, tcp_size));
2820150 }
```

94.12.53 kernel/net/udp.h



Si veda la sezione [93.23](#).

```
2830001 #ifndef _KERNEL_NET_UDP_H
2830002 #define _KERNEL_NET_UDP_H    1
2830003 //-----
2830004 #include <netinet/udp.h>
2830005 #include <kernel/net.h>
2830006 //-----
2830007 #define UDP_HEADER_SIZE    8
2830008 #define UDP_MAX_PACKET_SIZE  NET_IP_MAX_DATA_SIZE
2830009 #define UDP_MAX_DATA_SIZE   \
2830010     UDP_MAX_PACKET_SIZE-UDP_HEADER_SIZE
2830011 //-----
2830012 //
2830013 // UDP packet, for transmission.
2830014 //
2830015 typedef struct
2830016 {
2830017     struct udphdr header;
2830018     uint8_t data[UDP_MAX_DATA_SIZE];
2830019 } __attribute__((packed)) udp_packet_t;
2830020 //
2830021 // UDP pseudo header for checksum calculation.
2830022 //
2830023 typedef struct
2830024 {
2830025     in_addr_t saddr;
2830026     in_addr_t daddr;
2830027     uint8_t zero;
2830028     uint8_t protocol;
2830029     uint16_t length;
2830030 } __attribute__((packed)) udp_pseudo_header_t;
2830031 //-----
2830032 int udp_tx (h_port_t sport, h_port_t dport,
2830033             h_addr_t saddr, h_addr_t daddr,
2830034             const void *buffer, size_t size);
```

```
2830035 //-----  
2830036 #endif
```

94.12.54 kernel/net/udp/udp_tx.c



Si veda la sezione [93.23](#).

```
2840001 #include <kernel/net.h>  
2840002 #include <kernel/net/ip.h>  
2840003 #include <kernel/net/route.h>  
2840004 #include <kernel/net/udp.h>  
2840005 #include <sys/os32.h>  
2840006 #include <kernel/lib_k.h>  
2840007 #include <errno.h>  
2840008 #include <arpa/inet.h>  
2840009 #include <netinet/udp.h>  
2840010 //-----  
2840011 #define DEBUG 0  
2840012 //-----  
2840013 int  
2840014 udp_tx (h_port_t sport, h_port_t dport, h_addr_t saddr,  
2840015         h_addr_t daddr, const void *buffer, size_t size)  
2840016 {  
2840017     udp_packet_t packet;  
2840018     udp_pseudo_header_t pseudo;  
2840019     uint16_t checksum;  
2840020     size_t udp_size = size + sizeof (struct udphdr);  
2840021     //  
2840022     // Verify to have the source address: it is  
2840023     // necessary here for  
2840024     // the checksum calculation.  
2840025     //  
2840026     if (saddr == 0)  
2840027     {  
2840028         //  
2840029         // Default source address: get the source  
2840030         // address from the routing
```

```
2840031     // table, based on the destination.
2840032     //
2840033     saddr = route_remote_to_local (daddr);
2840034     if (saddr == ((h_addr_t) - 1))
2840035     {
2840036         errset (errno);
2840037         return (-1);
2840038     }
2840039 }
2840040 //
2840041 // Prepare the UDP packet.
2840042 //
2840043 packet.header.source = htons (sport);
2840044 packet.header.dest = htons (dport);
2840045 packet.header.len = htons (udp_size);
2840046 packet.header.check = 0;
2840047 memcpy (packet.data, buffer, size);
2840048 //
2840049 // Prepare the pseudo header.
2840050 //
2840051 pseudo.saddr = htonl (saddr);
2840052 pseudo.daddr = htonl (daddr);
2840053 pseudo.zero = 0;
2840054 pseudo.protocol = IPPROTO_UDP;
2840055 pseudo.length = htons (udp_size);
2840056 //
2840057 // Now set the header checksum.
2840058 //
2840059 checksum = ~(ip_checksum ((void *) &packet, udp_size,
2840060                          (void *) &pseudo,
2840061                          sizeof (udp_pseudo_header_t)));
2840062 if (checksum == 0)
2840063 {
2840064     checksum = 0xFFFF;
2840065 }
2840066 packet.header.check = htons (checksum);
2840067 //
```

```

2840068 // Send to the lower network level.
2840069 //
2840070 return (ip_tx
2840071         (saddr, daddr, (int) IPPROTO_UDP, &packet,
2840072         udp_size));
2840073 }

```

94.13 os32: «kernel/part.h»

Si veda la sezione [93.18](#).

```

2850001 #ifndef _KERNEL_PART_H
2850002 #define _KERNEL_PART_H    1
2850003
2850004
2850005 typedef struct
2850006 {
2850007     uint8_t active;           // boot indicator 0 or
2850008     // PART_ACTIVE
2850009     uint8_t h_start;        // head value for first sector
2850010     uint8_t s_start;        // sector value + cyl bits for
2850011     // first sector
2850012     uint8_t c_start;        // track value for first
2850013     // sector
2850014     uint8_t type; // partition type
2850015     uint8_t h_last;         // head value for last sector
2850016     uint8_t s_last;         // sector value + cyl bits for
2850017     // last sector
2850018     uint8_t c_last;         // track value for last sector
2850019     uint32_t l_start;        // logical first sector
2850020     uint32_t size;           // size of partition in
2850021     // sectors
2850022 } part_t;
2850023
2850024 #define PART_ACTIVE    0x80 // value for active
2850025                          // partition
2850026 #define PART_MAX      4    // number of entries

```

```

2850027 // in partition table
2850028 #define PART_TABLE_OFF 0x1BE // offset of part.
2850029 // table in boot
2850030 // sector
2850031
2850032 //
2850033 // Partition types.
2850034 //
2850035 #define PART_TYPE_NONE 0x00 // unused
2850036 // entry
2850037 #define PART_TYPE_NO_PART 0xFF // full device
2850038 #define PART_TYPE_MINIX 0x81 // Minix
2850039 // partition
2850040 // type
2850041 #define PART_TYPE_OLDMINIX 0x80 // Minix 1 old
2850042 // partition
2850043 // type
2850044 #define PART_TYPE_EXT 0x05 // extended
2850045 // partition
2850046
2850047
2850048 #endif

```

94.14 os32: «kernel/proc.h»

«

Si veda la sezione [93.20](#).

```

2860001 #ifndef _KERNEL_PROC_H
2860002 #define _KERNEL_PROC_H 1
2860003 //-----
2860004 #include <kernel/ibm_i386.h>
2860005 #include <kernel/dev.h>
2860006 #include <kernel/driver/tty.h>
2860007 #include <sys/types.h>
2860008 #include <sys/stat.h>

```

```
2860009 #include <kernel/fs.h>
2860010 #include <sys/os32.h>
2860011 #include <stddef.h>
2860012 #include <stdint.h>
2860013 #include <time.h>
2860014 //-----
2860015 #define CLOCK_FREQUENCY_DIVISOR \
2860016     (3579545/3/CLOCKS_PER_SEC) // [1]
2860017 //
2860018 // [1] Internal clock frequency is (3579545/3) Hz.
2860019 //     This value is divided by
2860020 //     CLOCK_FREQUENCY_DIVISOR, giving 100 Hz.
2860021 //     The divisor value is fixed, because the code
2860022 //     suppose that the clock frequency is 100 Hz!
2860023 //
2860024 //-----
2860025 #define PROC_EMPTY                0
2860026 #define PROC_CREATED              1
2860027 #define PROC_READY                2
2860028 #define PROC_RUNNING              3
2860029 #define PROC_SLEEPING            4
2860030 #define PROC_ZOMBIE              5
2860031 //-----
2860032 #define MAGIC_OS32_APPL    0x6F7333326170706CLL // [2]
2860033 //
2860034 // [2] os32appl
2860035 //
2860036 //-----
2860037 #define PROCESS_MAX    ((GDT_ITEMS/2)-1) // Process
2860038 // slots.
2860039 #define MAX_SIGNALS    31 // Max signal number.
2860040 //
2860041 typedef struct
2860042 {
2860043     pid_t ppid; // Parent PID.
2860044     pid_t pgrp; // Process group ID.
2860045     uid_t uid; // Real user ID
```

```
2860046 uid_t  eid;    // Effective user ID.
2860047 uid_t  suid;   // Saved user ID.
2860048 gid_t  gid;    // Real group ID
2860049 gid_t  egid;   // Effective group ID.
2860050 gid_t  sgid;   // Saved group ID.
2860051 dev_t  device_tty; // Controlling terminal.
2860052 char  path_cwd[PATH_MAX];
2860053 // Working directory path.
2860054 inode_t *inode_cwd; // Working directory inode.
2860055 int  umask;    // File creation mask.
2860056 unsigned long int sig_status; // Active signals.
2860057 unsigned long int sig_ignore; // Signals to be
2860058 // ignored.
2860059 uintptr_t sig_handler[MAX_SIGNALS]; // Opt. sig.
2860060 // handlers.
2860061 uintptr_t sig_handler_wrapper; // Special
2860062 // wrapper.
2860063 clock_t usage; // Clock ticks CPU time usage.
2860064 unsigned int status;
2860065 int  wakeup_events; // Wake up for something.
2860066 int  wakeup_signal; // Signal waited.
2860067 unsigned int wakeup_timer; // Seconds to wait
2860068 // for.
2860069 inode_t *wakeup_inode; // Inode waited.
2860070 sock_t *wakeup_sock; // Socket waited.
2860071 dev_t  wakeup_dev; // Device waited.
2860072 addr_t address_text;
2860073 size_t domain_text;
2860074 addr_t address_data;
2860075 size_t domain_data;
2860076 size_t domain_stack; // Included inside the data.
2860077 size_t extra_data; // Extra data for 'brk()'.
2860078 uint32_t sp;
2860079 int  ret;
2860080 char name[PATH_MAX];
2860081 fd_t  fd[FOPEN_MAX];
2860082 } proc_t;
```



```
2860083 //
2860084 extern proc_t proc_table[PROCESS_MAX];
2860085 //
2860086 // ATTENTION: THERE IS NO WAY TO KEEP THE STACK DOMAIN
2860087 // IN A DIFFERENT ADDRESS SPACE THAN THE
2860088 // OTHER DATA. There is a simple explanation
2860089 // for such limitation: A POINTER TO DATA
2860090 // INSIDE THE STACK, CANNOT BE REACHED IF IT
2860091 // IS NOT INSIDE THE SAME ADDRESS SPACE!
2860092 //
2860093 // For the same reason, data or stack,
2860094 // cannot be moved (shifted) down, when more
2860095 // space is needed, because previous
2860096 // pointers would not work! So, to develop
2860097 // an extensible 'malloc' area, such memory
2860098 // is to be placed *after* the stack, moving
2860099 // *all* the data segment if necessary.
2860100 //-----
2860101 extern pid_t proc_current;
2860102 extern uint32_t proc_stack_pointer;
2860103 extern uint16_t proc_stack_segment_selector;
2860104 extern unsigned int proc_loops_per_clock;
2860105 //-----
2860106 typedef struct
2860107 {
2860108     uint32_t filler0;
2860109     uint64_t magic;
2860110     uint32_t data_offset;
2860111     uint32_t etext;
2860112     uint32_t edata;
2860113     uint32_t ebss;
2860114     uint32_t ssize;
2860115 } header_t;
2860116 //-----
2860117 typedef struct
2860118 {
2860119     uint32_t eax;
```

```
2860120     uint32_t ecx;
2860121     uint32_t edx;
2860122     uint32_t ebx;
2860123     uint32_t ebp;
2860124     uint32_t esi;
2860125     uint32_t edi;
2860126     uint32_t ds;
2860127     uint32_t es;
2860128     uint32_t fs;
2860129     uint32_t gs;
2860130     uint32_t eip;
2860131     uint32_t cs;
2860132     uint32_t eflags;
2860133 } stack_t;
2860134 //-----
2860135 void proc_init (void);
2860136 void proc_timer_init (clock_t freq);
2860137 void proc_delay (void);
2860138 void proc_scheduler (void);
2860139 void sysroutine (uint32_t syscallnr, uint32_t msg_off,
2860140                 uint32_t msg_size);
2860141 //-----
2860142 void *ptr (pid_t pid, void *p);
2860143 proc_t *proc_reference (pid_t pid);
2860144 void proc_print (void);
2860145 //-----
2860146 int proc_sys_exec (pid_t pid, const char *path,
2860147                  unsigned int argc, char *arg_data,
2860148                  unsigned int envc, char *env_data);
2860149 //-----
2860150 void proc_dump_memory (pid_t pid, addr_t address,
2860151                      size_t size, char *name);
2860152 void proc_available (pid_t pid);
2860153 void proc_sch_net (void);
2860154 void proc_sch_signals (void);
2860155 void proc_sch_terminals (void);
2860156 void proc_sch_timers (void);
```

```
2860157 void proc_sig_chld (pid_t parent, int sig);
2860158 void proc_sig_cont (pid_t pid, int sig);
2860159 void proc_sig_core (pid_t pid, int sig);
2860160 void proc_sig_handler (pid_t pid, int sig);
2860161 int proc_sig_ignore (pid_t pid, int sig);
2860162 void proc_sig_off (pid_t pid, int sig);
2860163 void proc_sig_on (pid_t pid, int sig);
2860164 int proc_sig_status (pid_t pid, int sig);
2860165 void proc_sig_stop (pid_t pid, int sig);
2860166 void proc_sig_term (pid_t pid, int sig);
2860167
2860168 void proc_wakeup_pipe_read (inode_t * inode);
2860169 void proc_wakeup_pipe_write (inode_t * inode);
2860170 void proc_wakeup_terminal (void);
2860171
2860172 #endif
```

94.14.1	kernel/proc/proc_available.c	1692
94.14.2	kernel/proc/proc_dump_memory.c	1693
94.14.3	kernel/proc/proc_init.c	1695
94.14.4	kernel/proc/proc_print.c	1701
94.14.5	kernel/proc/proc_public.c	1705
94.14.6	kernel/proc/proc_reference.c	1705
94.14.7	kernel/proc/proc_sch_net.c	1706
94.14.8	kernel/proc/proc_sch_signals.c	1709
94.14.9	kernel/proc/proc_sch_terminals.c	1710
94.14.10	kernel/proc/proc_sch_timers.c	1721
94.14.11	kernel/proc/proc_scheduler.c	1722
94.14.12	kernel/proc/proc_sig_chld.c	1728

94.14.13	kernel/proc/proc_sig_cont.c	1730
94.14.14	kernel/proc/proc_sig_core.c	1731
94.14.15	kernel/proc/proc_sig_handler.c	1733
94.14.16	kernel/proc/proc_sig_ignore.c	1740
94.14.17	kernel/proc/proc_sig_off.c	1740
94.14.18	kernel/proc/proc_sig_on.c	1741
94.14.19	kernel/proc/proc_sig_status.c	1741
94.14.20	kernel/proc/proc_sig_stop.c	1742
94.14.21	kernel/proc/proc_sig_term.c	1742
94.14.22	kernel/proc/proc_sys_exec.c	1744
94.14.23	kernel/proc/proc_timer_init.c	1767
94.14.24	kernel/proc/proc_wakeup_pipe_read.c	1768
94.14.25	kernel/proc/proc_wakeup_pipe_write.c	1769
94.14.26	kernel/proc/proc_wakeup_terminal.c	1769
94.14.27	kernel/proc/ptr.c	1771
94.14.28	kernel/proc/sysroutine.c	1771

94.14.1 kernel/proc/proc_available.c



Si veda la sezione [93.20.1](#).

```
2870001 #include <kernel/proc.h>
2870002 //-----
2870003 void
2870004 proc_available (pid_t pid)
2870005 {
```

```
2870006     proc_table[pid].ppid = (pid_t) - 1;
2870007     proc_table[pid].pgrp = (pid_t) - 1;
2870008     proc_table[pid].uid = (uid_t) - 1;
2870009     proc_table[pid].euid = (uid_t) - 1;
2870010     proc_table[pid].suid = (uid_t) - 1;
2870011     proc_table[pid].gid = (gid_t) - 1;
2870012     proc_table[pid].egid = (gid_t) - 1;
2870013     proc_table[pid].sgid = (gid_t) - 1;
2870014     proc_table[pid].sig_status = 0;
2870015     proc_table[pid].sig_ignore = 0;
2870016     proc_table[pid].usage = (clock_t) 0;
2870017     proc_table[pid].status = PROC_EMPTY;
2870018     proc_table[pid].wakeup_events = 0;
2870019     proc_table[pid].wakeup_signal = 0;
2870020     proc_table[pid].wakeup_timer = 0;
2870021     proc_table[pid].wakeup_inode = NULL;
2870022     proc_table[pid].address_text = (addr_t) 0;
2870023     proc_table[pid].domain_text = (size_t) 0;
2870024     proc_table[pid].address_data = (addr_t) 0;
2870025     proc_table[pid].domain_data = (size_t) 0;
2870026     proc_table[pid].domain_stack = (size_t) 0;
2870027     proc_table[pid].extra_data = (size_t) 0;
2870028     proc_table[pid].sp = (uint32_t) 0;
2870029     proc_table[pid].ret = 0;
2870030     proc_table[pid].inode_cwd = NULL;
2870031     proc_table[pid].path_cwd[0] = 0;
2870032     proc_table[pid].umask = 0;
2870033     proc_table[pid].name[0] = 0;
2870034 }
```

94.14.2 kernel/proc/proc_dump_memory.c

Si veda la sezione [93.20.2](#).

```
2880001     #include <kernel/proc.h>
2880002     #include <fcntl.h>
2880003     #include <kernel/lib_s.h>
```

```
2880004 #include <kernel/lib_k.h>
2880005 //-----
2880006 void
2880007 proc_dump_memory (pid_t pid, addr_t address,
2880008                   size_t size, char *name)
2880009 {
2880010     int fdn;
2880011     char buffer[BUFSIZ];
2880012     ssize_t size_written;
2880013     ssize_t size_written_total;
2880014     ssize_t size_read;
2880015     ssize_t size_read_total;
2880016     ssize_t size_total = 0;
2880017     //
2880018     // Dump the code segment to disk.
2880019     //
2880020     fdn =
2880021         s_open (pid, name, (O_WRONLY | O_CREAT | O_TRUNC),
2880022               (mode_t) (S_IFREG | 00644));
2880023     if (fdn < 0)
2880024     {
2880025         //
2880026         // There is a problem: just let it go.
2880027         //
2880028         k_perror (NULL);
2880029         return;
2880030     }
2880031     //
2880032     // Read the memory and write it to disk.
2880033     //
2880034     for (size_read = 0, size_read_total = 0;
2880035          size_read_total < size_total;
2880036          size_read_total += size_read, address += size_read)
2880037     {
2880038         size_read = dev_io ((pid_t) 0, DEV_MEM, DEV_READ,
2880039                            (off_t) address, buffer,
2880040                            (size_t) BUFSIZ, NULL);
```

```
2880041 //
2880042 for (size_written = 0, size_written_total = 0;
2880043     size_written_total < size_read;
2880044     size_written_total += size_written)
2880045 {
2880046     size_written = s_write
2880047         (pid, fdn,
2880048          &buffer
2880049          [size_written_total],
2880050          (size_t) (size_read - size_written_total));
2880051 //
2880052 if (size_written < 0)
2880053 {
2880054     s_close (pid, fdn);
2880055     return;
2880056 }
2880057 }
2880058 }
2880059 s_close (pid, fdn);
2880060 }
```

94.14.3 kernel/proc/proc_init.c

Si veda la sezione [93.20.3](#).

```
2890001 #include <kernel/ibm_i386.h>
2890002 #include <kernel/proc.h>
2890003 #include <kernel/lib_k.h>
2890004 #include <kernel/lib_s.h>
2890005 #include <string.h>
2890006 #include <kernel/multiboot.h>
2890007 #include <kernel/dm.h>
2890008 //-----
2890009 extern uint32_t _k_start;
2890010 extern uint32_t _k_end;
2890011 extern uint32_t _k_text_end;
2890012 extern uint32_t _k_data_end;
```

```
2890013 extern uint32_t _k_bss_end;
2890014 extern uint32_t _k_stack_top;
2890015 extern uint32_t _k_stack_bottom;
2890016 //-----
2890017 void
2890018 proc_init (void)
2890019 {
2890020     pid_t pid;
2890021     int fdn;          // File descriptor index;
2890022     inode_t *inode;
2890023     sb_t *sb;
2890024     clock_t time_start;
2890025     clock_t time_now;
2890026     clock_t time_elapsed;
2890027     unsigned long long int count;
2890028     uintptr_t stack_top = (uintptr_t) &_k_stack_top;
2890029     uintptr_t stack_bottom = (uintptr_t) &_k_stack_bottom;
2890030     int sig;
2890031     //
2890032     // Set up the GDT table.
2890033     //
2890034     gdt ();
2890035     //
2890036     // Set up timer.
2890037     //
2890038     proc_timer_init (CLOCKS_PER_SEC);
2890039     //
2890040     // Disable external interrupt.
2890041     //
2890042     cli ();
2890043     //
2890044     // Set up the IDT table.
2890045     //
2890046     idt ();
2890047     //
2890048     // Set all memory reference to some invalid data.
2890049     //
```



```
2890050     for (pid = 0; pid < PROCESS_MAX; pid++)
2890051     {
2890052         proc_available (pid);
2890053     }
2890054     //
2890055     // Set up the process table with the kernel.
2890056     //
2890057     proc_table[0].ppid = 0;
2890058     proc_table[0].pgrp = 0;
2890059     proc_table[0].uid = 0;
2890060     proc_table[0].euid = 0;
2890061     proc_table[0].suid = 0;
2890062     proc_table[0].gid = 0;
2890063     proc_table[0].egid = 0;
2890064     proc_table[0].sgid = 0;
2890065     proc_table[0].device_tty = DEV_UNDEFINED;
2890066     proc_table[0].sig_status = 0;
2890067     proc_table[0].sig_ignore = 0;
2890068     proc_table[0].usage = 0;
2890069     proc_table[0].status = PROC_RUNNING;
2890070     proc_table[0].wakeup_events = 0;
2890071     proc_table[0].wakeup_signal = 0;
2890072     proc_table[0].wakeup_timer = 0;
2890073     proc_table[0].wakeup_inode = NULL;
2890074     proc_table[0].address_text = 0; // [1]
2890075     proc_table[0].domain_text = (size_t) (&k_end); // [2]
2890076     proc_table[0].address_data = 0; // [2]
2890077     proc_table[0].domain_data = (size_t) 0; // [2]
2890078     proc_table[0].domain_stack =
2890079         (size_t) (stack_bottom - stack_top);
2890080     proc_table[0].extra_data = (size_t) 0;
2890081     proc_table[0].sp = 0; // [3]
2890082     proc_table[0].ret = 0;
2890083     proc_table[0].umask = 0022; // Default umask.
2890084     strncpy (proc_table[0].path_cwd, "/", PATH_MAX);
2890085     strncpy (proc_table[0].name, "os32 kernel", PATH_MAX);
2890086     //
```

```
2890087 // [1] The kernel text starts at 0x100000, that is,
2890088 // 1 Mibyte,
2890089 // but the code expect to start at that address,
2890090 // just like
2890091 // the space from address 0 to 0xFFFFF is anyway
2890092 // part
2890093 // of the kernel. If the kernel is forked, as it
2890094 // happens
2890095 // when the 'init' process is to be run, it is
2890096 // necessary
2890097 // to consider part of the kernel all the addresses
2890098 // starting
2890099 // from zero.
2890100 //
2890101 // [2] The kernel is a unique block, where text and
2890102 // data live
2890103 // together.
2890104 //
2890105 // [3] The saved stack pointer location will be set
2890106 // at the
2890107 // next interrupt, or system call. That is why the
2890108 // kernel
2890109 // must send a null system call at the beginning of
2890110 // its
2890111 // work.
2890112 // -----
2890113 //
2890114 // Ensure to have a terminated string.
2890115 //
2890116 proc_table[0].name[PATH_MAX - 1] = 0;
2890117 //
2890118 // Reset file descriptors.
2890119 //
2890120 for (fdn = 0; fdn < OPEN_MAX; fdn++)
2890121 {
2890122     proc_table[0].fd[fdn].fl_flags = 0;
2890123     proc_table[0].fd[fdn].fd_flags = 0;
```

```
2890124     proc_table[0].fd[fdn].file = NULL;
2890125     }
2890126     //
2890127     // Reset 'sig_handler[]'.
2890128     //
2890129     for (sig = 0; sig < MAX_SIGNALS; sig++)
2890130     {
2890131         proc_table[0].sig_handler[sig] = (uintptr_t) NULL;
2890132     }
2890133     //
2890134     // Allocate memory for the kernel.
2890135     //
2890136     // The BIOS data area (BDA) and extra BIOS at the
2890137     // bottom
2890138     // of 640 Kibyte, is already, formally, included
2890139     // inside the
2890140     // kernel.
2890141     //
2890142     // The allocation for data has no effect here,
2890143     // because it is
2890144     // the same as the text.
2890145     //
2890146     mb_alloc (proc_table[0].address_text,
2890147              proc_table[0].domain_text);
2890148     mb_alloc (proc_table[0].address_data,
2890149              proc_table[0].domain_data);
2890150     //
2890151     // Enable and disable hardware interrupts (IRQ).
2890152     //
2890153     irq_on (0);    // timer.
2890154     irq_on (1);    // keyboard
2890155     irq_on (2);    // PIC2: keep it ON! [4]
2890156     irq_on (3);    //
2890157     irq_on (4);    //
2890158     irq_on (5);    //
2890159     irq_on (6);    //
2890160     irq_on (7);    //
```

```
2890161     irq_on (8);    //
2890162     irq_on (9);    //
2890163     irq_on (10);   //
2890164     irq_on (11);   //
2890165     irq_on (12);   //
2890166     irq_on (13);   //
2890167     irq_on (14);   //
2890168     irq_on (15);   //
2890169     //
2890170     // [4] IRQ 2 must be let working, because inside the
2890171     // file
2890172     // 'kernel/ibm_i386/isr_s', the IRQ 2 is used to
2890173     // reset
2890174     // properly PIC 1, so that, the IRQ from 8 to 15
2890175     // will
2890176     // reset only PIC 2.
2890177     // The problem is that any PIC must be reset exactly
2890178     // once, otherwise the system will lock.
2890179     //
2890180     //
2890181     // External interrupts activation.
2890182     //
2890183     sti ();
2890184     //
2890185     // Calculate how many times can be executed the
2890186     // following loop. This data will be used by
2890187     // 'k_sleep()', if the clock pulse does not work
2890188     // for some reason.
2890189     //
2890190     time_elapsed = 0;
2890191     count = 0;
2890192     time_start = s_clock ((pid_t) 0);
2890193     for (; time_elapsed < 10; count++)
2890194     {
2890195         time_now = s_clock ((pid_t) 0);
2890196         time_elapsed = time_now - time_start;
2890197     }
```

```
2890198     proc_loops_per_clock = count / 10;
2890199     //
2890200     // Set up data-memory devices:
2890201     // it works only after enabling interrupts.
2890202     //
2890203     dm_init ();
2890204     //
2890205     // Mount root file system.
2890206     //
2890207     inode = NULL;
2890208     sb = sb_mount (DEV_DM02, &inode, MOUNT_DEFAULT);
2890209     if (sb == NULL || inode == NULL)
2890210     {
2890211         k_perror
2890212             ("Kernel panic: cannot mount " "root file system:");
2890213         k_exit ();
2890214     }
2890215     //
2890216     // Add the inode to the process table item for the
2890217     // kernel.
2890218     //
2890219     proc_table[0].inode_cwd = inode;           // Root fs
2890220     // inode.
2890221 }
```

94.14.4 kernel/proc/proc_print.c

Si veda la sezione [93.20.4](#).

```
2900001 #include <sys/os32.h>
2900002 #include <kernel/proc.h>
2900003 #include <kernel/lib_k.h>
2900004 //-----
2900005 void
2900006 proc_print (void)
2900007 {
2900008     pid_t pid;
```



```
2900009 char stat;
2900010 unsigned int min;
2900011 unsigned int sec;
2900012 //
2900013 //
2900014 //
2900015 k_printf
2900016 ("pp p pg T * 0x1000 "
2900017 "D * 0x1000 stack \n"
2900018 "id id rp tty uid euid suid usage s addr size "
2900019 "addr size pointer name\n");
2900020 //
2900021 //
2900022 //
2900023 for (pid = 0; pid < PROCESS_MAX; pid++)
2900024 {
2900025     if (proc_table[pid].status > 0)
2900026     {
2900027         switch (proc_table[pid].status)
2900028         {
2900029             case PROC_EMPTY:
2900030                 stat = '-';
2900031                 break;
2900032             case PROC_CREATED:
2900033                 stat = 'c';
2900034                 break;
2900035             case PROC_READY:
2900036                 stat = 'r';
2900037                 break;
2900038             case PROC_RUNNING:
2900039                 stat = 'R';
2900040                 break;
2900041             case PROC_SLEEPING:
2900042                 stat = 's';
2900043                 break;
2900044             case PROC_ZOMBIE:
2900045                 stat = 'z';
```

```
2900046         break;
2900047     default:
2900048         stat = '?';
2900049         break;
2900050     }
2900051     //
2900052     min =
2900053         ((proc_table[pid].usage / CLOCKS_PER_SEC) / 60);
2900054     sec =
2900055         ((proc_table[pid].usage / CLOCKS_PER_SEC) % 60);
2900056     //
2900057     // Addresses and sizes are multiple of 4096
2900058     // (0x1000);
2900059     // for the stack pointer is shown only the
2900060     // last five
2900061     // hexadecimal digits.
2900062     //
2900063     if (proc_table[pid].domain_data > 0)
2900064     {
2900065         k_printf
2900066             ("%2i %2i %2i %04x %4i %4i "
2900067              "%4i %02i.%02i %c "
2900068              "%05x %04x %05x %04x %07x %12s\n",
2900069              (unsigned int) proc_table[pid].ppid,
2900070              (unsigned int) pid,
2900071              (unsigned int) proc_table[pid].pgrp,
2900072              (unsigned int) proc_table[pid].device_tty,
2900073              (unsigned int) proc_table[pid].uid,
2900074              (unsigned int) proc_table[pid].euid,
2900075              (unsigned int) proc_table[pid].suid,
2900076              min, sec, stat,
2900077              (unsigned int) proc_table[pid].address_text
2900078              / MEM_BLOCK_SIZE,
2900079              (unsigned int) proc_table[pid].domain_text
2900080              / MEM_BLOCK_SIZE,
2900081              (unsigned int) proc_table[pid].address_data
2900082              / MEM_BLOCK_SIZE,
```

```
2900083         (unsigned int) (proc_table[pid].domain_data
2900084                     +
2900085                     proc_table[pid].extra_data)
2900086         / MEM_BLOCK_SIZE,
2900087         (unsigned int) proc_table[pid].sp,
2900088         proc_table[pid].name);
2900089     }
2900090     else
2900091     {
2900092         k_printf
2900093         ("%2i %2i %2i %04x %4i %4i %4i "
2900094          "%02i.%02i %c "
2900095          "%05x %04x %05x %04x %07x %12s\n",
2900096          (unsigned int) proc_table[pid].ppid,
2900097          (unsigned int) pid,
2900098          (unsigned int) proc_table[pid].pgrp,
2900099          (unsigned int) proc_table[pid].device_tty,
2900100          (unsigned int) proc_table[pid].uid,
2900101          (unsigned int) proc_table[pid].euid,
2900102          (unsigned int) proc_table[pid].suid,
2900103          min, sec, stat,
2900104          (unsigned int) proc_table[pid].address_text
2900105          / MEM_BLOCK_SIZE,
2900106          (unsigned int) (proc_table[pid].domain_text
2900107                          +
2900108                          proc_table[pid].extra_data)
2900109          / MEM_BLOCK_SIZE,
2900110          (unsigned int) proc_table[pid].address_data
2900111          / MEM_BLOCK_SIZE,
2900112          (unsigned int) proc_table[pid].domain_data
2900113          / MEM_BLOCK_SIZE,
2900114          (unsigned int) proc_table[pid].sp,
2900115          proc_table[pid].name);
2900116     }
2900117 }
2900118 }
```



```
2900119 }  
}
```

94.14.5 kernel/proc/proc_public.c

Si veda la sezione [93.20.5](#).

```
2910001 #include <kernel/proc.h>  
2910002 //-----  
2910003 proc_t proc_table[PROCESS_MAX];  
2910004 pid_t proc_current = 0;  
2910005 uint16_t proc_stack_segment_selector = 24;  
2910006 uint32_t proc_stack_pointer;  
2910007 unsigned int proc_loops_per_clock;
```

94.14.6 kernel/proc/proc_reference.c

Si veda la sezione [93.20.5](#).

```
2920001 #include <kernel/proc.h>  
2920002 //-----  
2920003 proc_t *  
2920004 proc_reference (pid_t pid)  
2920005 {  
2920006     if (pid >= 0 && pid < PROCESS_MAX)  
2920007     {  
2920008         return (&proc_table[pid]);  
2920009     }  
2920010     else  
2920011     {  
2920012         return (NULL);  
2920013     }  
2920014 }
```

94.14.7 kernel/proc/proc_sch_net.c

<<

Si veda la sezione [93.20.6](#).

```
2930001 #include <kernel/proc.h>
2930002 #include <kernel/lib_k.h>
2930003 #include <kernel/lib_s.h>
2930004 #include <kernel/driver/nic/ne2k.h>
2930005 #include <kernel/net.h>
2930006 #include <kernel/driver/pci.h>
2930007 //-----
2930008 #define DEBUG 0
2930009 //-----
2930010 void
2930011 proc_sch_net (void)
2930012 {
2930013     int n;          // NET table index.
2930014     int counter = 0;
2930015     int pid;
2930016     int tcp_wake_up;
2930017     //
2930018     // Do what there is to do with network interfaces,
2930019     // in particular get
2930020     // received frames, into the
2930021     // 'net_table[n]...buffer[f]' table.
2930022     //
2930023     for (n = 0; n < NET_MAX_DEVICES; n++)
2930024     {
2930025         if (net_table[n].type == NET_DEV_ETH_NE2K)
2930026         {
2930027             //
2930028             // The function 'ne2k_isr()' will call also
2930029             // 'ne2k_rx()'
2930030             // that is responsible for placing Ethernet
2930031             // frames inside
2930032             // 'ethernet_table[eth].frame_info[f]'.
2930033             //
2930034             ne2k_isr (net_table[n].ethernet.base_io);
```

```
2930035     }
2930036     else if (net_table[n].type == NET_DEV_LOOPBACK)
2930037     {
2930038         //
2930039         // Packets should be already inside the
2930040         // packet buffer table.
2930041         //
2930042         ;
2930043     }
2930044 }
2930045 //
2930046 // Do something with received Ethernet frames.
2930047 //
2930048 counter = net_rx ();
2930049 //
2930050 if (DEBUG)
2930051 {
2930052     if (counter)
2930053     {
2930054         k_printf ("+ %i packet(s) received\n", counter);
2930055     }
2930056 }
2930057 //
2930058 // Do something with TCP connections.
2930059 //
2930060 tcp_wake_up = tcp ();
2930061 //
2930062 // Wake up sleeping processes.
2930063 //
2930064 if (counter || tcp_wake_up == TCP_TRY_READ)
2930065 {
2930066     //
2930067     // Wake up sleeping processes, waiting to read
2930068     // from a socket.
2930069     //
2930070     if (DEBUG)
2930071     {
```

```
2930072         k_printf
2930073             ("wake up processes, waiting "
2930074             "for a socket!\n");
2930075     }
2930076     for (pid = 1; pid < PROCESS_MAX; pid++)
2930077     {
2930078         if (proc_table[pid].status == PROC_SLEEPING
2930079             && (proc_table[pid].wakeup_events
2930080                 & WAKEUP_EVENT_SOCKET_READ))
2930081             {
2930082                 proc_table[pid].wakeup_events = 0;
2930083                 proc_table[pid].status = PROC_READY;
2930084             }
2930085     }
2930086 }
2930087 //
2930088 if (tcp_wake_up == TCP_TRY_WRITE)
2930089 {
2930090     //
2930091     // Wake up sleeping processes, waiting to write
2930092     // to a socket.
2930093     //
2930094     for (pid = 1; pid < PROCESS_MAX; pid++)
2930095     {
2930096         if (proc_table[pid].status == PROC_SLEEPING
2930097             && (proc_table[pid].wakeup_events
2930098                 & WAKEUP_EVENT_SOCKET_WRITE))
2930099             {
2930100                 proc_table[pid].wakeup_events = 0;
2930101                 proc_table[pid].status = PROC_READY;
2930102             }
2930103     }
2930104 }
2930105 }
```

94.14.8 kernel/proc/proc_sch_signals.c



Si veda la sezione [93.20.7](#).

```
2940001 #include <kernel/proc.h>
2940002 //-----
2940003 void
2940004 proc_sch_signals (void)
2940005 {
2940006     pid_t pid;
2940007     //
2940008     // The PID scan starts from 1: you will not send
2940009     // signals to the
2940010     // kernel!
2940011     //
2940012     for (pid = 1; pid < PROCESS_MAX; pid++)
2940013     {
2940014         proc_sig_term (pid, SIGHUP);
2940015         proc_sig_term (pid, SIGINT);
2940016         proc_sig_core (pid, SIGQUIT);
2940017         proc_sig_core (pid, SIGILL);
2940018         proc_sig_core (pid, SIGABRT);
2940019         proc_sig_core (pid, SIGFPE);
2940020         proc_sig_term (pid, SIGKILL);
2940021         proc_sig_core (pid, SIGSEGV);
2940022         proc_sig_term (pid, SIGPIPE);
2940023         proc_sig_term (pid, SIGALRM);
2940024         proc_sig_term (pid, SIGTERM);
2940025         proc_sig_term (pid, SIGUSR1);
2940026         proc_sig_term (pid, SIGUSR2);
2940027         proc_sig_chld (pid, SIGCHLD);
2940028         proc_sig_cont (pid, SIGCONT);
2940029         proc_sig_stop (pid, SIGSTOP);
2940030         proc_sig_stop (pid, SIGTSTP);
2940031         proc_sig_stop (pid, SIGTTIN);
2940032         proc_sig_stop (pid, SIGTTOU);
2940033     }
2940034 }
```

94.14.9 kernel/proc/proc_sch_terminals.c

<<

Si veda la sezione [93.20.8](#).

```
2950001 #include <kernel/proc.h>
2950002 #include <kernel/lib_k.h>
2950003 #include <kernel/lib_s.h>
2950004 #include <kernel/driver/kbd.h>
2950005 #include <termios.h>
2950006 #include <limits.h>
2950007 //-----
2950008 void
2950009 proc_sch_terminals (void)
2950010 {
2950011     pid_t pid;
2950012     pid_t pid_sub;
2950013     unsigned char key;
2950014     tty_t *tty;
2950015     dev_t device;
2950016     int k;          // Reverse count killed character.
2950017     int overflow = 0;    // True if input is too much.
2950018     //
2950019     // Try to read a key from console keyboard buffer
2950020     // (only consoles
2950021     // are available). Variable 'kbd' is external and
2950022     // comes from
2950023     // 'kernel/kbd.h'.
2950024     //
2950025     key = kbd.key;
2950026     if (key == 0)
2950027     {
2950028         //
2950029         // No key is ready on the keyboard buffer: just
2950030         // return.
2950031         //
2950032         return;
2950033     }
2950034     else
```

```
2950035     {
2950036         //
2950037         // A key was pressed and it will be processed.
2950038         // Currently, just remove from the keyboard
2950039         // buffer.
2950040         kbd.key = 0;
2950041     }
2950042     //
2950043     // A key is available. Find the currently active
2950044     // console.
2950045     //
2950046     device = tty_console ((dev_t) 0);
2950047     tty = tty_reference (device);
2950048     if (tty == NULL)
2950049     {
2950050         k_printf
2950051             ("kernel alert: console device "
2950052              "0x%04x not found!\n", device);
2950053         //
2950054         // Will send the typed character to the first
2950055         // terminal!
2950056         //
2950057         tty = tty_reference ((dev_t) 0);
2950058     }
2950059     //
2950060     // Verify if it is a control key that must be
2950061     // handled before
2950062     // entering the canonical input line.
2950063     //
2950064     // Check for a console switch key combination.
2950065     //
2950066     if (key == 0x11)        // [Ctrl Q] -> DC1 ->
2950067         // console0.
2950068     {
2950069         tty_console (DEV_CONSOLE0);        // Switch.
2950070         return;
2950071     }
```

```
2950072 else if (key == 0x12) // [Ctrl R] -> DC2 ->
2950073     // console1.
2950074     {
2950075         tty_console (DEV_CONSOLE1);           // Switch.
2950076         return;
2950077     }
2950078 else if (key == 0x13) // [Ctrl S] -> DC3 ->
2950079     // console2.
2950080     {
2950081         tty_console (DEV_CONSOLE2);           // Switch.
2950082         return;
2950083     }
2950084 else if (key == 0x14) // [Ctrl T] -> DC4 ->
2950085     // console3.
2950086     {
2950087         tty_console (DEV_CONSOLE3);           // Switch.
2950088         return;
2950089     }
2950090 // -----
2950091 // Only canonical input line is available: ICANON!
2950092 // -----
2950093 //
2950094 // Might be necessary to send a signal to all
2950095 // processes with the
2950096 // same control terminal, excluded the kernel (0)
2950097 // and 'init' (1).
2950098 // Such control keys are not passed to the
2950099 // applications, or are
2950100 // replaced with zero.
2950101 //
2950102 // Please note that this a simplified solution,
2950103 // because the signal
2950104 // should reach only the foreground process of the
2950105 // group. For that
2950106 // reason, only che [Ctrl C] is taken into
2950107 // consideration, because
2950108 // processes can ignore the signal 'SIGINT'.
```



```
2950109 //
2950110 if (tty->pgrp != 0)
2950111 {
2950112 //
2950113 // There is a process group for that terminal.
2950114 //
2950115 if (tty->attr.c_cc[VINTR]
2950116     && key == tty->attr.c_cc[VINTR])
2950117 {
2950118     if (tty->attr.c_iflag & IGNBRK)
2950119     {
2950120 //
2950121 // No break!
2950122 //
2950123     return;
2950124     }
2950125 //
2950126 if (tty->attr.c_iflag & BRKINT)
2950127 {
2950128     if (tty->attr.c_lflag & ISIG)
2950129     {
2950130         for (pid = 2; pid < PROCESS_MAX; pid++)
2950131         {
2950132             if (proc_table[pid].pgrp == tty->pgrp)
2950133             {
2950134 //
2950135 // Should find only final
2950136 // process/processes.
2950137 // So, if there is a son
2950138 // process with the
2950139 // same process group, the
2950140 // signal is not
2950141 // sent!
2950142 //
2950143             for (pid_sub = 2;
2950144                 pid_sub < PROCESS_MAX;
2950145                 pid_sub++)
```

```
2950146         {
2950147             if ((proc_table[pid_sub].ppid
2950148                 == pid)
2950149                 &&
2950150                 (proc_table[pid_sub].pgrp
2950151                 == tty->pgrp))
2950152             {
2950153                 //
2950154                 // 'pid_sub' is a
2950155                 // son of
2950156                 // 'pid' and is part
2950157                 // of the
2950158                 // same process
2950159                 // group. 'pid_sub'
2950160                 // is candidate for
2950161                 // kill.
2950162                 //
2950163                 break;
2950164             }
2950165         }
2950166     //
2950167     if (pid_sub >= PROCESS_MAX)
2950168     {
2950169         //
2950170         // There is no son for
2950171         // 'pid', sorry.
2950172         //
2950173         s_kill ((pid_t) 0, pid,
2950174                SIGINT);
2950175         //
2950176         // No more scan.
2950177         //
2950178         break;
2950179     }
2950180 }
2950181 }
2950182 }
```

```
2950183         //
2950184         // Just reset input line and return.
2950185         //
2950186         tty->status = TTY_INPUT_LINE_EDITING;
2950187         tty->lpr = 0;
2950188         tty->lpw = 0;
2950189         tty->line[0] = 0;
2950190         //
2950191         return;
2950192     }
2950193     //
2950194     // Replace the INTR character with zero.
2950195     //
2950196     key = 0;
2950197 }
2950198 }
2950199 //
2950200 // Check if something is to be ignored.
2950201 //
2950202 if (key == '\r' && (tty->attr.c_iflag & IGNCR))
2950203 {
2950204     return;
2950205 }
2950206 //
2950207 // Check if something is to be replaced, before
2950208 // editing.
2950209 //
2950210 if (key == '\n' && (tty->attr.c_iflag & INLCR))
2950211 {
2950212     key = '\r';
2950213 }
2950214 //
2950215 if (key == '\r' && (tty->attr.c_iflag & ICRNL))
2950216 {
2950217     key = '\n';
2950218 }
2950219 //
```

```
2950220 // Edit the canonical input line.
2950221 // Input is accepted only if status is ok.
2950222 //
2950223 if (tty->status == TTY_INPUT_LINE_EDITING)
2950224 {
2950225     //
2950226     // Fix internal line positions.
2950227     //
2950228     if (tty->lpw < 0)
2950229     {
2950230         tty->lpw = 0;
2950231     }
2950232     if (tty->lpw >= MAX_CANON)
2950233     {
2950234         tty->lpw = MAX_CANON - 1;
2950235         overflow = 1; // Too much input!
2950236     }
2950237     if (tty->lpr < 0)
2950238     {
2950239         tty->lpr = 0;
2950240     }
2950241     if (tty->lpr > tty->lpw)
2950242     {
2950243         tty->lpr = tty->lpw;
2950244     }
2950245     //
2950246     // Check input key.
2950247     //
2950248     if (key == '\0')
2950249     {
2950250         //
2950251         // A INTR replaced into zero.
2950252         //
2950253         tty->line[tty->lpw] = key;
2950254         tty->status = TTY_INPUT_LINE_CLOSED;
2950255         proc_wakeup_terminal ();
2950256     }
```

```
2950257     else if (key == '\n')
2950258     {
2950259         tty->line[tty->lpw] = key;
2950260         tty->status = TTY_INPUT_LINE_CLOSED;
2950261         proc_wakeup_terminal ();
2950262     }
2950263     else if (tty->attr.c_cc[VEOF]
2950264             && key == tty->attr.c_cc[VEOF])
2950265     {
2950266         //
2950267         // EOF is not included inside the line: just
2950268         // replace the
2950269         // with zero.
2950270         //
2950271         key = 0;
2950272         tty->line[tty->lpw] = key;
2950273         tty->status = TTY_INPUT_LINE_CLOSED;
2950274         proc_wakeup_terminal ();
2950275     }
2950276     else if (tty->attr.c_cc[VEOL]
2950277             && key == tty->attr.c_cc[VEOL])
2950278     {
2950279         tty->line[tty->lpw] = key;
2950280         tty->status = TTY_INPUT_LINE_CLOSED;
2950281         proc_wakeup_terminal ();
2950282     }
2950283     else if (tty->attr.c_cc[VERASE]
2950284             && key == tty->attr.c_cc[VERASE])
2950285     {
2950286         //
2950287         // Save how many characters to be killed, if
2950288         // echo is
2950289         // enabled.
2950290         //
2950291         if (overflow)
2950292         {
2950293             k = tty->lpw + 1;
```

```
2950294         //
2950295         // The 'tty->lpw' was already reduced.
2950296         //
2950297     }
2950298     else
2950299     {
2950300         k = tty->lpw;
2950301         //
2950302         // Reduce write index: if it is less
2950303         // than zero, it will
2950304         // be fixed.
2950305         //
2950306         tty->lpw--;
2950307     }
2950308 }
2950309 else if (tty->attr.c_cc[VKILL]
2950310         && key == tty->attr.c_cc[VKILL])
2950311 {
2950312     //
2950313     // Save how many characters to be killed, if
2950314     // echo is
2950315     // enabled.
2950316     //
2950317     if (overflow)
2950318     {
2950319         k = tty->lpw + 1;
2950320     }
2950321     else
2950322     {
2950323         k = tty->lpw;
2950324     }
2950325     //
2950326     // Reset input line.
2950327     //
2950328     tty->lpw = 0;
2950329     tty->lpr = 0;
2950330     tty->line[0] = 0;
```

```
2950331     }
2950332     else
2950333     {
2950334         tty->line[tty->lpw] = key;
2950335         tty->lpw++;
2950336     }
2950337     //
2950338     // Echo.
2950339     //
2950340     if (!(tty->attr.c_lflag & ECHO))
2950341     {
2950342         //
2950343         // No echo. But NL might be echoed anyway.
2950344         //
2950345         if (key == '\n' && (tty->attr.c_lflag & ECHONL))
2950346         {
2950347             dev_io ((pid_t) 0, tty->device,
2950348                 DEV_WRITE, (off_t) 0, &key,
2950349                 (size_t) 1, NULL);
2950350         }
2950351         //
2950352         return;
2950353     }
2950354     //
2950355     // The echo is requested.
2950356     //
2950357     if (key == 0)
2950358     {
2950359         //
2950360         // There is nothing to echo.
2950361         //
2950362         ;
2950363     }
2950364     else if (tty->attr.c_cc[VERASE]
2950365             && key == tty->attr.c_cc[VERASE])
2950366     {
2950367         if (tty->attr.c_lflag & ECHOE)
```

```
2950368     {
2950369         if (k > 0)
2950370         {
2950371             dev_io ((pid_t) 0, tty->device,
2950372                    DEV_WRITE, (off_t) 0,
2950373                    "\b \b", (size_t) 3, NULL);
2950374         }
2950375     }
2950376     else
2950377     {
2950378         dev_io ((pid_t) 0, tty->device,
2950379                DEV_WRITE, (off_t) 0, &key,
2950380                (size_t) 1, NULL);
2950381     }
2950382 }
2950383 else if (tty->attr.c_cc[VKILL]
2950384          && key == tty->attr.c_cc[VKILL])
2950385     {
2950386         if (tty->attr.c_lflag & ECHOK)
2950387         {
2950388             for (; k > 0; k--)
2950389             {
2950390                 dev_io ((pid_t) 0, tty->device,
2950391                        DEV_WRITE, (off_t) 0,
2950392                        "\b \b", (size_t) 3, NULL);
2950393             }
2950394         }
2950395     }
2950396 else if (key == '\n')
2950397     {
2950398         if (tty->attr.c_lflag & ECHONL)
2950399         {
2950400             dev_io ((pid_t) 0, tty->device,
2950401                    DEV_WRITE, (off_t) 0, &key,
2950402                    (size_t) 1, NULL);
2950403         }
2950404     }
```



```
2950405     else
2950406     {
2950407         //
2950408         // If there was an overflow, the last
2950409         // character is
2950410         // overwriting the last position, so a back
2950411         // space
2950412         // is printed.
2950413         //
2950414         if (overflow)
2950415         {
2950416             dev_io ((pid_t) 0, tty->device,
2950417                    DEV_WRITE, (off_t) 0, "\b",
2950418                    (size_t) 1, NULL);
2950419         }
2950420         //
2950421         // Now show the input character.
2950422         //
2950423         dev_io ((pid_t) 0, tty->device, DEV_WRITE,
2950424                (off_t) 0, &key, (size_t) 1, NULL);
2950425     }
2950426 }
2950427 }
```

94.14.10 kernel/proc/proc_sch_timers.c

Si veda la sezione [93.20.9](#).

```
2960001 #include <kernel/proc.h>
2960002 #include <kernel/lib_k.h>
2960003 #include <kernel/lib_s.h>
2960004 //-----
2960005 void
2960006 proc_sch_timers (void)
2960007 {
2960008     static unsigned long long int previous_time;
2960009     unsigned long long int current_time;
```

```
2960010 unsigned int pid;
2960011 current_time = s_time ((pid_t) 0, NULL);
2960012 if (previous_time != current_time)
2960013     {
2960014         for (pid = 0; pid < PROCESS_MAX; pid++)
2960015             {
2960016                 if ((proc_table[pid].wakeup_events &
2960017                     WAKEUP_EVENT_TIMER)
2960018                     && (proc_table[pid].status == PROC_SLEEPING)
2960019                     && (proc_table[pid].wakeup_timer > 0))
2960020                     {
2960021                         proc_table[pid].wakeup_timer--;
2960022                         if (proc_table[pid].wakeup_timer == 0)
2960023                             {
2960024                                 proc_table[pid].status = PROC_READY;
2960025                             }
2960026                     }
2960027             }
2960028     }
2960029     previous_time = current_time;
2960030 }
```

94.14.11 kernel/proc/proc_scheduler.c



Si veda la sezione [93.20.10](#).

```
2970001 #include <kernel/proc.h>
2970002 #include <kernel/lib_k.h>
2970003 #include <kernel/lib_s.h>
2970004 #include <kernel/net.h>
2970005 #include <stdint.h>
2970006 //-----
2970007 #define DEBUG 1
2970008 //-----
2970009 extern uint32_t _ksp;
2970010 extern uint32_t proc_stack_pointer;
2970011 extern uint16_t proc_stack_segment_selector;
```

```
2970012 extern pid_t proc_current;
2970013 //-----
2970014 void
2970015 proc_scheduler (void)
2970016 {
2970017     pid_t prev;
2970018     pid_t next;
2970019     addr_t stack_top;
2970020     addr_t stack_bottom;
2970021     uint32_t saved_stack_pointer = proc_stack_pointer;
2970022     //
2970023     static unsigned long long int previous_clock;
2970024     unsigned long long int current_clock;
2970025     //
2970026     // Check the current stack size.
2970027     //
2970028     if (proc_table[proc_current].domain_data == 0)
2970029     {
2970030         stack_bottom = proc_table[proc_current].domain_text;
2970031     }
2970032     else
2970033     {
2970034         stack_bottom = proc_table[proc_current].domain_data;
2970035     }
2970036     //
2970037     stack_top =
2970038         stack_bottom - proc_table[proc_current].domain_stack;
2970039     //
2970040     // Check if the process has broken data with the
2970041     // stack,
2970042     // or if it is near the end of its domain.
2970043     //
2970044     if (proc_stack_pointer <= stack_top)
2970045     {
2970046         //
2970047         // The stack overlaped the other data!
2970048         //
```

```
2970049     k_printf
2970050     ("[%s] Kernel alert: the stack of process %i "
2970051     "is grown beyond the allowed space! "
2970052     "The process "
2970053     "is closed. Stack top is %i, "
2970054     "stack pointer is %i.\n",
2970055     __FILE__, (int) proc_current, (int) stack_top,
2970056     (int) proc_stack_pointer);
2970057     //
2970058     // The process is terminated badly.
2970059     //
2970060     s__exit (proc_current, -1);
2970061 }
2970062 else if (proc_stack_pointer < (stack_top + 1024))
2970063 {
2970064     //
2970065     // There is only 1 Kibyte and the stack is
2970066     // finished!
2970067     //
2970068     k_printf
2970069     ("[%s] Kernel alert: the stack of process %i "
2970070     "is near the end of the allowed space! "
2970071     "It remains only %i byte and it will "
2970072     "overwrite other data!\n", __FILE__,
2970073     (int) proc_current,
2970074     (int) (proc_stack_pointer - stack_top));
2970075 }
2970076 //
2970077 // Save previous PID. Variable 'proc_current' is
2970078 // extern.
2970079 //
2970080 prev = proc_current;
2970081 //
2970082 // Take care of networking.
2970083 //
2970084 proc_sch_net ();
2970085 //
```

```
2970086 // Take care of sleeping processes: wake up if
2970087 // sleeping time
2970088 // elapsed.
2970089 //
2970090 proc_sch_timers ();
2970091 //
2970092 // Take care of pending signals.
2970093 //
2970094 proc_sch_signals ();
2970095 //
2970096 // Take care input from terminals.
2970097 //
2970098 proc_sch_terminals ();
2970099 //
2970100 // Update the CPU time usage.
2970101 //
2970102 current_clock = s_clock ((pid_t) 0);
2970103 proc_table[prev].usage += current_clock - previous_clock;
2970104 previous_clock = current_clock;
2970105 //
2970106 // Check stack pointer changes, made probably by
2970107 // 'proc_sig_handler()' called from
2970108 // 'proc_sch_signals()'.
2970109 //
2970110 if (DEBUG)
2970111     {
2970112         if (saved_stack_pointer != proc_stack_pointer)
2970113             {
2970114                 k_printf
2970115                     ("%s] pid %i, ESP from %i to %i.\n",
2970116                     __FILE__, proc_current,
2970117                     saved_stack_pointer, proc_stack_pointer);
2970118             }
2970119     }
2970120 //
2970121 // Scan for a next process.
2970122 //
```

```
2970123   for (next = prev + 1; next != prev; next++)
2970124       {
2970125           if (next >= PROCESS_MAX)
2970126               {
2970127                 next = -1;    // At the next loop, 'next'
2970128                 // will be zero.
2970129                 continue;
2970130               }
2970131           //
2970132           if (proc_table[next].status == PROC_EMPTY)
2970133               {
2970134                 continue;
2970135               }
2970136           else if (proc_table[next].status == PROC_CREATED)
2970137               {
2970138                 continue;
2970139               }
2970140           else if (proc_table[next].status == PROC_READY)
2970141               {
2970142                 if (proc_table[prev].status == PROC_RUNNING)
2970143                     {
2970144                       proc_table[prev].status = PROC_READY;
2970145                     }
2970146                 //
2970147                 proc_table[prev].sp = proc_stack_pointer;
2970148                 proc_table[next].status = PROC_RUNNING;
2970149                 proc_table[next].ret = 0;
2970150                 //
2970151                 proc_current = next;
2970152                 proc_stack_segment_selector
2970153                     = gdt_pid_to_segment_data (next) * 8;
2970154                 proc_stack_pointer = proc_table[next].sp;
2970155                 break;
2970156               }
2970157           else if (proc_table[next].status == PROC_RUNNING)
2970158               {
2970159                 if (proc_table[prev].status == PROC_RUNNING)
```

```
2970160         {
2970161             k_printf ("Kernel alert: process %i "
2970162                     "and %i \"running\"!\n",
2970163                     prev, next);
2970164             proc_table[prev].status = PROC_READY;
2970165         }
2970166         //
2970167         proc_table[prev].sp = proc_stack_pointer;
2970168         proc_table[next].status = PROC_RUNNING;
2970169         proc_table[next].ret = 0;
2970170         //
2970171         proc_current = next;
2970172         proc_stack_segment_selector
2970173             = gdt_pid_to_segment_data (next) * 8;
2970174         proc_stack_pointer = proc_table[next].sp;
2970175         break;
2970176     }
2970177     else if (proc_table[next].status == PROC_SLEEPING)
2970178     {
2970179         continue;
2970180     }
2970181     else if (proc_table[next].status == PROC_ZOMBIE)
2970182     {
2970183         continue;
2970184     }
2970185 }
2970186 //
2970187 // Check again if the next process is set to
2970188 // running, otherwise set
2970189 // the kernel to such value!
2970190 //
2970191 if (proc_table[next].status != PROC_RUNNING)
2970192 {
2970193     proc_table[0].status = PROC_RUNNING;
2970194     proc_current = 0;
2970195     proc_stack_segment_selector
2970196         = gdt_pid_to_segment_data (0) * 8;
```

```
2970197     proc_stack_pointer = proc_table[0].sp;
2970198     }
2970199     //
2970200     // Save kernel stack pointer.
2970201     //
2970202     _ksp = proc_table[0].sp;
2970203 }
```

94.14.12 kernel/proc/proc_sig_chld.c

«

Si veda la sezione [93.20.11](#).

```
2980001 #include <kernel/proc.h>
2980002 //-----
2980003 // At the moment, the SIGCHLD is handled only per
2980004 // default: no other handler is taken into
2980005 // consideration.
2980006 //-----
2980007 void
2980008 proc_sig_chld (pid_t parent, int sig)
2980009 {
2980010     pid_t child;
2980011     //
2980012     // Please note that 'sig' should be SIGCHLD and
2980013     // nothing else.
2980014     // So, the following test, means to verify if the
2980015     // parent process
2980016     // has received a SIGCHLD already.
2980017     //
2980018     if (proc_sig_status (parent, sig))
2980019     {
2980020         if ((!proc_sig_ignore (parent, sig))
2980021             && (proc_table[parent].status ==
2980022                 PROC_SLEEPING)
2980023             && (proc_table[parent].wakeup_events &
2980024                 WAKEUP_EVENT_SIGNAL)
2980025             && (proc_table[parent].wakeup_signal == sig))
```



```
2980026     {
2980027         //
2980028         // The signal is not ignored from the parent
2980029         // process;
2980030         // the parent process is sleeping;
2980031         // the parent process is waiting for a
2980032         // signal;
2980033         // the parent process is waiting for current
2980034         // signal.
2980035         // So, just wake it up.
2980036         //
2980037         proc_table[parent].status = PROC_READY;
2980038         proc_table[parent].wakeup_events = 0;
2980039         proc_table[parent].wakeup_signal = 0;
2980040     }
2980041     else
2980042     {
2980043         //
2980044         // All other cases, means to remove all dead
2980045         // children.
2980046         //
2980047         for (child = 1; child < PROCESS_MAX; child++)
2980048             {
2980049                 if (proc_table[child].ppid == parent
2980050                     && proc_table[child].status ==
2980051                         PROC_ZOMBIE)
2980052                     {
2980053                         proc_available (child);
2980054                     }
2980055             }
2980056     }
2980057     proc_sig_off (parent, sig);
2980058 }
2980059 }
```

94.14.13 kernel/proc/proc_sig_cont.c



Si veda la sezione [93.20.12](#).

```
2990001 #include <kernel/proc.h>
2990002 //-----
2990003 void
2990004 proc_sig_cont (pid_t pid, int sig)
2990005 {
2990006     //
2990007     // The value for argument 'sig' should be SIGCONT.
2990008     //
2990009     if (proc_sig_status (pid, sig))
2990010     {
2990011         if (proc_sig_ignore (pid, sig))
2990012         {
2990013             proc_sig_off (pid, sig);
2990014         }
2990015         else
2990016         {
2990017             if (proc_table[pid].sig_handler[sig] !=
2990018                 (uintptr_t) NULL)
2990019             {
2990020                 proc_sig_handler (pid, sig);
2990021             }
2990022             else
2990023             {
2990024                 proc_table[pid].status = PROC_READY;
2990025                 proc_sig_off (pid, sig);
2990026             }
2990027         }
2990028     }
2990029 }
```

94.14.14 kernel/proc/proc_sig_core.c



Si veda la sezione [93.20.13](#).

```
3000001 #include <kernel/proc.h>
3000002 #include <kernel/lib_s.h>
3000003 //-----
3000004 void
3000005 proc_sig_core (pid_t pid, int sig)
3000006 {
3000007     addr_t address_text;
3000008     addr_t address_data;
3000009     size_t domain_text;
3000010     size_t domain_data;
3000011     size_t extra_data;
3000012     //
3000013     if (proc_sig_status (pid, sig))
3000014     {
3000015         if (proc_sig_ignore (pid, sig))
3000016         {
3000017             proc_sig_off (pid, sig);
3000018         }
3000019     else
3000020     {
3000021         if (proc_table[pid].sig_handler[sig] !=
3000022             (uintptr_t) NULL)
3000023         {
3000024             proc_sig_handler (pid, sig);
3000025         }
3000026     else
3000027     {
3000028         //
3000029         // Save process addresses and sizes
3000030         // (might be useful if
3000031         // we want to try to exit the process
3000032         // before core dump.
3000033         //
3000034         address_text = proc_table[pid].address_text;
```

```
300035 address_data = proc_table[pid].address_data;
300036 domain_text = proc_table[pid].domain_text;
300037 domain_data = proc_table[pid].domain_data;
300038 extra_data = proc_table[pid].extra_data;
300039 //
300040 // Core dump: the process who formally
300041 // writes the file
300042 // is the terminating one.
300043 //
300044 if (domain_data == 0)
300045 {
300046     proc_dump_memory (pid, address_text,
300047                      domain_text +
300048                      extra_data, "core");
300049 }
300050 else
300051 {
300052     proc_dump_memory (pid, address_text,
300053                      domain_text,
300054                      "core.text");
300055     proc_dump_memory (pid, address_data,
300056                      domain_data +
300057                      extra_data,
300058                      "core.data");
300059 }
300060 //
300061 // The signal, translated to negative,
300062 // is returned (but
300063 // the effective value received by the
300064 // application will
300065 // be cutted, leaving only the low 8
300066 // bit).
300067 //
300068 s__exit (pid, -sig);
300069 }
300070 }
300071 }
```

3000072	}
---------	---

94.14.15 kernel/proc/proc_sig_handler.c

Si veda la sezione [93.20.14](#).

```
3010001 #include <kernel/proc.h>
3010002 #include <kernel/lib_s.h>
3010003 #include <kernel/lib_k.h>
3010004 #include <stdint.h>
3010005 //-----
3010006 #define DEBUG 0
3010007 //-----
3010008 void
3010009 proc_sig_handler (pid_t pid, int sig)
3010010 {
3010011     addr_t addr_data_top;
3010012     addr_t addr_stack_pointer;
3010013     uint32_t old_eip;
3010014     uint32_t old_cs;
3010015     uint32_t old_eflags;
3010016     //
3010017     // Stack frames.
3010018     //
3010019     struct
3010020     {
3010021         uint32_t eax;
3010022         uint32_t ecx;
3010023         uint32_t edx;
3010024         uint32_t ebx;
3010025         uint32_t ebp;
3010026         uint32_t esi;
3010027         uint32_t edi;
3010028         uint32_t ds;
3010029         uint32_t es;
3010030         uint32_t fs;
3010031         uint32_t gs;
```

```
3010032     uint32_t eip;
3010033     uint32_t cs;
3010034     uint32_t eflags;
3010035 } *old;
3010036 //
3010037 struct
3010038 {
3010039     uint32_t eax;
3010040     uint32_t ecx;
3010041     uint32_t edx;
3010042     uint32_t ebx;
3010043     uint32_t ebp;
3010044     uint32_t esi;
3010045     uint32_t edi;
3010046     uint32_t ds;
3010047     uint32_t es;
3010048     uint32_t fs;
3010049     uint32_t gs;
3010050     uint32_t wrapper;
3010051     uint32_t cs;
3010052     uint32_t eflags;
3010053     uint32_t handler;
3010054     uint32_t signal;
3010055     uint32_t eip;
3010056 } *new;
3010057 //
3010058 // First of all, this function can act only for a
3010059 // process that
3010060 // is not *just* interrupted. That is, if
3010061 // 'proc_current' is
3010062 // equal to 'pid', nothing is to be done yet: it
3010063 // will be done
3010064 // only when it will be in pause.
3010065 //
3010066 if (pid == proc_current)
3010067     {
3010068         //
```

```
3010069      // Nothing to do yet.
3010070      //
3010071      return;
3010072  }
3010073  //
3010074  // Check if there is a function to run.
3010075  //
3010076  if (proc_table[pid].sig_handler[sig] == (uintptr_t) NULL)
3010077  {
3010078      //
3010079      // Nothing to do.
3010080      //
3010081      return;
3010082  }
3010083  //
3010084  // Tell something for debugging.
3010085  //
3010086  if (DEBUG)
3010087  {
3010088      k_printf ("%s(%i, %i)", __func__, (int) pid, sig);
3010089  }
3010090  //
3010091  // Calculate the absolute stack section address,
3010092  // from the
3010093  // kernel point of view.
3010094  //
3010095  if (proc_table[pid].domain_data == 0)
3010096  {
3010097      addr_data_top = proc_table[pid].address_text;
3010098  }
3010099  else
3010100  {
3010101      addr_data_top = proc_table[pid].address_data;
3010102  }
3010103  //
3010104  // Then calculate the effective stack pointer
3010105  // address.
```

```
3010106 // We are considering only process that are not
3010107 // currently interrupted, so the stack pointer is
3010108 // taken from 'proc_table[pid].sp'.
3010109 //
3010110 addr_stack_pointer = addr_data_top + proc_table[pid].sp;
3010111 //
3010112 // Currently the process stack is as it follows.
3010113 // The address inside 'addr_stack_pointer' is
3010114 // corresponding
3010115 // to the saved EAX value.
3010116 //
3010117 // pushl %eflags #
3010118 // pushl %cs # from the interrupt
3010119 // pushl %eip #
3010120 // -----
3010121 // pushl %gs
3010122 // pushl %fs
3010123 // pushl %es
3010124 // pushl %ds
3010125 // pushl %edi
3010126 // pushl %esi
3010127 // pushl %ebp
3010128 // pushl %ebx
3010129 // pushl %edx
3010130 // pushl %ecx
3010131 // pushl %eax
3010132 //
3010133 // Need to insert the call to the handler function.
3010134 // The process stack should become this way:
3010135 //
3010136 // pushl %eip [0]
3010137 // pushl <signal> [1]
3010138 // pushl <handler> [1]
3010139 // pushl %eflags [2]
3010140 // pushl %cs [2]
3010141 // pushl <wrapper> [2]
3010142 // -----
```



```
3010143 // pushl %gs
3010144 // pushl %fs
3010145 // pushl %es
3010146 // pushl %ds
3010147 // pushl %edi
3010148 // pushl %esi
3010149 // pushl %ebp
3010150 // pushl %ebx
3010151 // pushl %edx
3010152 // pushl %ecx
3010153 // pushl %eax
3010154 //
3010155 // [0] Back from the original interrupt.
3010156 //
3010157 // [1] Arguments of the wrapper functions, that will
3010158 // call the
3010159 // signal handler, and then will return to the
3010160 // address at
3010161 // [1].
3010162 //
3010163 // [2] Modified so that the IRET instruction will
3010164 // return
3010165 // to the begin of the wrapper function, that will
3010166 // call the true signal handler, and will return
3010167 // at [1].
3010168 //
3010169 // Now set the pointer to the old and new frame,
3010170 // updating the stack pointer address.
3010171 //
3010172 old = (void *) addr_stack_pointer;
3010173 addr_stack_pointer -= 12; // Three more
3010174 // elements.
3010175 new = (void *) addr_stack_pointer;
3010176 //
3010177 // Verify if the code segment was correctly found.
3010178 //
3010179 if (DEBUG)
```

```
3010180     {
3010181     k_printf
3010182         ("%s] EAX:%04x ECX:%04x EDX:%04x "
3010183         "EBX:%04x EBP:%04x "
3010184         "ESI:%04x EDI:%04x DS:%04x ES:%04x "
3010185         "FS:%04x GS:%04x "
3010186         "EIP:%04x CS:%04x EFLAGS:%04x\n", __FILE__,
3010187         (int) old->eax, (int) old->ecx,
3010188         (int) old->edx, (int) old->ebx,
3010189         (int) old->ebp, (int) old->esi,
3010190         (int) old->edi, (int) old->ds, (int) old->es,
3010191         (int) old->fs, (int) old->gs, (int) old->eip,
3010192         (int) old->cs, (int) old->eflags);
3010193     }
3010194     //
3010195     // Move data, to arrange the new stack. The order
3010196     // does
3010197     // matter, as the new frame overwrites the old one.
3010198     //
3010199     new->eax = old->eax;
3010200     new->ecx = old->ecx;
3010201     new->edx = old->edx;
3010202     new->ebx = old->ebx;
3010203     new->ebp = old->ebp;
3010204     new->esi = old->esi;
3010205     new->edi = old->edi;
3010206     new->ds = old->ds;
3010207     new->es = old->es;
3010208     new->fs = old->fs;
3010209     new->gs = old->gs;
3010210     //
3010211     old_eflags = old->eflags;
3010212     old_cs = old->cs;
3010213     old_eip = old->eip;
3010214     //
3010215     new->wrapper = proc_table[pid].sig_handler_wrapper;
3010216     new->cs = old_cs;
```

```
3010217     new->eflags = old_eflags;
3010218     new->handler = proc_table[pid].sig_handler[sig];
3010219     new->signal = sig;
3010220     new->eip = old_eip;
3010221     //
3010222     // Tell what is changed inside the stack.
3010223     //
3010224     if (DEBUG)
3010225     {
3010226         k_printf
3010227         ("[%s] EAX:%04x ECX:%04x EDX:%04x "
3010228          "EBX:%04x EBP:%04x "
3010229          "ESI:%04x EDI:%04x DS:%04x ES:%04x "
3010230          "FS:%04x GS:%04x "
3010231          "wrapper:%04x CS:%04x EFLAGS:%04x "
3010232          "handler:%04x "
3010233          "signal:$04x EIP:%04x\n", __FILE__,
3010234          (int) new->eax, (int) new->ecx,
3010235          (int) new->edx, (int) new->ebx,
3010236          (int) new->ebp, (int) new->esi,
3010237          (int) new->edi, (int) new->ds, (int) new->es,
3010238          (int) new->fs, (int) new->gs,
3010239          (int) new->wrapper, (int) new->cs,
3010240          (int) new->eflags, (int) new->handler,
3010241          (int) new->signal, (int) new->eip);
3010242     }
3010243     //
3010244     // Change the stack pointer of the process, as it
3010245     // was increased.
3010246     //
3010247     proc_table[pid].sp = addr_stack_pointer - addr_data_top;
3010248     //
3010249     // Reset the signal handler, as in traditional Unix,
3010250     // with
3010251     // all the consequences that such implementation
3010252     // will give.
3010253     //
```

```

3010254     proc_table[pid].sig_handler[sig] = (uintptr_t) NULL;
3010255     proc_sig_off (pid, sig);
3010256     //
3010257     // Wake up the process if it is sleeping.
3010258     //
3010259     proc_table[pid].wakeup_events = 0;
3010260     proc_table[pid].status = PROC_READY;
3010261 }

```

94.14.16 kernel/proc/proc_sig_ignore.c

«

Si veda la sezione [93.20.15](#).

```

3020001 #include <kernel/proc.h>
3020002 //-----
3020003 int
3020004 proc_sig_ignore (pid_t pid, int sig)
3020005 {
3020006     unsigned long int flag = 1L << (sig - 1);
3020007     if (proc_table[pid].sig_ignore & flag)
3020008     {
3020009         return (1);
3020010     }
3020011     else
3020012     {
3020013         return (0);
3020014     }
3020015 }

```

94.14.17 kernel/proc/proc_sig_off.c

«

Si veda la sezione [93.20.17](#).

```

3030001 #include <kernel/proc.h>
3030002 //-----
3030003 void
3030004 proc_sig_off (pid_t pid, int sig)

```

```
3030005 {
3030006     unsigned long int flag = 1L << (sig - 1);
3030007     proc_table[pid].sig_status ^= flag;
3030008 }
```

94.14.18 kernel/proc/proc_sig_on.c

Si veda la sezione [93.20.17](#).

```
3040001 #include <kernel/proc.h>
3040002 //-----
3040003 void
3040004 proc_sig_on (pid_t pid, int sig)
3040005 {
3040006     unsigned long int flag = 1L << (sig - 1);
3040007     proc_table[pid].sig_status |= flag;
3040008 }
```

94.14.19 kernel/proc/proc_sig_status.c

Si veda la sezione [93.20.18](#).

```
3050001 #include <kernel/proc.h>
3050002 //-----
3050003 int
3050004 proc_sig_status (pid_t pid, int sig)
3050005 {
3050006     unsigned long int flag = 1L << (sig - 1);
3050007     if (proc_table[pid].sig_status & flag)
3050008     {
3050009         return (1);
3050010     }
3050011     else
3050012     {
3050013         return (0);
3050014     }
3050015 }
```

94.14.20 kernel/proc/proc_sig_stop.c



Si veda la sezione [93.20.19](#).

```
3060001 #include <kernel/proc.h>
3060002 //-----
3060003 void
3060004 proc_sig_stop (pid_t pid, int sig)
3060005 {
3060006     if (proc_sig_status (pid, sig))
3060007     {
3060008         if (proc_sig_ignore (pid, sig) && !(sig == SIGSTOP))
3060009         {
3060010             proc_sig_off (pid, sig);
3060011         }
3060012     else
3060013     {
3060014         if ((proc_table[pid].sig_handler[sig] !=
3060015             (uintptr_t) NULL) && (sig != SIGSTOP))
3060016         {
3060017             proc_sig_handler (pid, sig);
3060018         }
3060019     else
3060020     {
3060021         proc_table[pid].status = PROC_SLEEPING;
3060022         proc_table[pid].ret = -sig;
3060023         proc_sig_off (pid, sig);
3060024     }
3060025 }
3060026 }
3060027 }
```

94.14.21 kernel/proc/proc_sig_term.c



Si veda la sezione [93.20.20](#).

```
3070001 #include <kernel/proc.h>
3070002 #include <kernel/lib_s.h>
```

```
3070003 #include <kernel/lib_k.h>
3070004 //-----
3070005 void
3070006 proc_sig_term (pid_t pid, int sig)
3070007 {
3070008     if (proc_sig_status (pid, sig))
3070009     {
3070010         if (proc_sig_ignore (pid, sig) && !(sig == SIGKILL))
3070011         {
3070012             proc_sig_off (pid, sig);
3070013         }
3070014     else
3070015     {
3070016         if ((proc_table[pid].sig_handler[sig] !=
3070017             (uintptr_t) NULL) && (sig != SIGKILL))
3070018         {
3070019             proc_sig_handler (pid, sig);
3070020         }
3070021     else
3070022     {
3070023         //
3070024         // The signal, translated to negative,
3070025         // is returned (but
3070026         // the effective value received by the
3070027         // application will
3070028         // be cutted, leaving only the low 8
3070029         // bit).
3070030         //
3070031         s__exit (pid, -sig);
3070032     }
3070033 }
3070034 }
3070035 }
```

94.14.22 kernel/proc/proc_sys_exec.c



Si veda la sezione [93.20.21](#).

```
3080001 #include <kernel/ibm_i386.h>
3080002 #include <kernel/proc.h>
3080003 #include <errno.h>
3080004 #include <fcntl.h>
3080005 #include <kernel/lib_s.h>
3080006 #include <kernel/lib_k.h>
3080007 //-----
3080008 #define DEBUG 0
3080009 //-----
3080010 int
3080011 proc_sys_exec (pid_t pid, const char *path,
3080012               unsigned int argc, char *arg_data,
3080013               unsigned int envc, char *env_data)
3080014 {
3080015     unsigned int i;
3080016     unsigned int j;
3080017     char *arg;
3080018     char *env;
3080019     char *envp[ARG_MAX / 16];
3080020     char *argv[ARG_MAX / 16];
3080021     size_t size;
3080022     size_t arg_data_size;
3080023     size_t env_data_size;
3080024     unsigned int p_off;
3080025     inode_t *inode;
3080026     ssize_t size_read;
3080027     header_t header;
3080028     uint32_t new_sp;
3080029     uint32_t envp_address;
3080030     uint32_t argv_address;
3080031     addr_t allocated_text;
3080032     addr_t allocated_data;
3080033     addr_t stack_location;           // real stack
3080034     // location.
```



```
3080035     size_t process_domain_text;
3080036     size_t process_domain_data;
3080037     size_t process_domain_stack;
3080038     addr_t previous_address_text;
3080039     addr_t previous_address_data;
3080040     size_t previous_domain_text;
3080041     size_t previous_domain_data;
3080042     size_t previous_domain_stack;
3080043     size_t previous_extra_data;
3080044     uint32_t segment_text;           // Segment descriptors
3080045     uint32_t segment_data;         // inside 32 bit int.
3080046     char buffer[MEM_BLOCK_SIZE];
3080047     uint32_t stack_element;
3080048     off_t off_inode;
3080049     addr_t memory_start;
3080050     int status;
3080051     pid_t extra;
3080052     int proc_count;
3080053     file_t *file;
3080054     int fdn;
3080055     dev_t device;
3080056     int eof;
3080057     int sig;
3080058     //
3080059     // Check for limits.
3080060     //
3080061     if (argc > (ARG_MAX / 16) || envc > (ARG_MAX / 16))
3080062     {
3080063         errset (ENOMEM);
3080064         return (-1);
3080065     }
3080066     //
3080067     // Scan arguments to calculate the full size and the
3080068     // relative
3080069     // pointers. The final size is rounded to 4, for the
3080070     // stack.
3080071     //
```

```
3080072 arg = arg_data;
3080073 for (i = 0, j = 0; i < argc; i++)
3080074 {
3080075     argv[i] = (char *) j;    // Relative pointer
3080076     // inside the
3080077     // 'arg_data'.
3080078     size = strlen (arg);
3080079     arg += size + 1;
3080080     j += size + 1;
3080081 }
3080082 arg_data_size = j;
3080083 if (arg_data_size % 2)
3080084 {
3080085     arg_data_size++;
3080086 }
3080087 if (arg_data_size % 4)
3080088 {
3080089     arg_data_size += 2;
3080090 }
3080091 //
3080092 // Scan environment variables to calculate the full
3080093 // size and the
3080094 // relative pointers. The final size is rounded to
3080095 // 4, for the stack.
3080096 //
3080097 env = env_data;
3080098 for (i = 0, j = 0; i < envc; i++)
3080099 {
3080100     envp[i] = (char *) j;    // Relative pointer
3080101     // inside the
3080102     // 'env_data'.
3080103     size = strlen (env);
3080104     env += size + 1;
3080105     j += size + 1;
3080106 }
3080107 env_data_size = j;
3080108 if (env_data_size % 2)
```

```
3080109     {
3080110         env_data_size++;
3080111     }
3080112     if (env_data_size % 4)
3080113     {
3080114         env_data_size += 2;
3080115     }
3080116     //
3080117     // Read the inode related to the executable file
3080118     // name.
3080119     // Function path_inode() includes the inode get
3080120     // procedure.
3080121     //
3080122     inode = path_inode (pid, path);
3080123     if (inode == NULL)
3080124     {
3080125         errset (ENOENT); // No such file or directory.
3080126         return (-1);
3080127     }
3080128     //
3080129     // Check for permissions.
3080130     //
3080131     status =
3080132         inode_check (inode, S_IFREG, 5,
3080133                     proc_table[pid].euid,
3080134                     proc_table[pid].egid);
3080135     if (status != 0)
3080136     {
3080137         //
3080138         // File is not of a valid type or permission are
3080139         // not
3080140         // sufficient: release the executable file inode
3080141         // and return with an error.
3080142         //
3080143         inode_put (inode);
3080144         errset (EACCES); // Permission denied.
3080145         return (-1);
```

```
3080146     }
3080147     //
3080148     // Read the header from the executable file.
3080149     //
3080150     size_read =
3080151         inode_file_read (inode, (off_t) 0, &header,
3080152                         (sizeof header), &eof);
3080153     if (size_read != (sizeof header))
3080154     {
3080155         //
3080156         // The file is shorter than the executable
3080157         // header, so, it isn't
3080158         // an executable: release the file inode and
3080159         // return with an
3080160         // error.
3080161         //
3080162         inode_put (inode);
3080163         errset (ENOEXEC);
3080164         return (-1);
3080165     }
3080166     //
3080167     // Size read is ok.
3080168     //
3080169     if (header.magic != MAGIC_OS32_APPL)
3080170     {
3080171         //
3080172         // The header does not have the expected magic
3080173         // numbers, so,
3080174         // it isn't a valid executable: release the file
3080175         // inode and
3080176         // return with an error.
3080177         //
3080178         inode_put (inode);
3080179         errset (ENOEXEC);
3080180         return (-1);        // This is not a valid
3080181         // executable!
3080182     }
```

```
3080183 //
3080184 // Calculate code size.
3080185 //
3080186 if (header.data_offset == 0)
3080187 {
3080188     process_domain_text = header.ebss + header.ssize;
3080189 }
3080190 else
3080191 {
3080192     process_domain_text = header.data_offset;
3080193 }
3080194 //
3080195 if (process_domain_text % 4096)
3080196 {
3080197     process_domain_text =
3080198         (((process_domain_text / 4096) + 1) * 4096);
3080199 }
3080200 //
3080201 // Calculate data size, including stack, that cannot
3080202 // stay alone!
3080203 //
3080204 process_domain_stack = header.ssize;
3080205 //
3080206 if (header.data_offset == 0)
3080207 {
3080208     process_domain_data = 0;
3080209 }
3080210 else
3080211 {
3080212     process_domain_data = (header.ebss + header.ssize);
3080213 }
3080214 //
3080215 if (process_domain_data % 4096)
3080216 {
3080217     process_domain_data =
3080218         (((process_domain_data / 4096) + 1) * 4096);
3080219 }
```

```
3080220 //
3080221 // Place the new stack pointer to the bottom of the
3080222 // data area:
3080223 // the stack pointer is relative to the data area,
3080224 // so the last
3080225 // relative position is equal to the size.
3080226 //
3080227 if (header.data_offset == 0)
3080228 {
3080229     new_sp = process_domain_text;
3080230 }
3080231 else
3080232 {
3080233     new_sp = process_domain_data;
3080234 }
3080235 //
3080236 // Allocate memory: code and data.
3080237 //
3080238 allocated_text = mb_alloc_size (process_domain_text);
3080239 //
3080240 if (allocated_text == 0)
3080241 {
3080242     //
3080243     // The program instructions (code segment)
3080244     // cannot be loaded
3080245     // into memory: release the executable file
3080246     // inode and return
3080247     // with an error.
3080248     //
3080249     inode_put (inode);
3080250     errset (ENOMEM); // Not enough space.
3080251     return (-1);
3080252 }
3080253 else if (DEBUG)
3080254 {
3080255     k_printf ("%s:%i:mb_alloc_size(%zi)", __FILE__,
3080256              __LINE__,
```

```
3080257         (unsigned int) process_domain_text);
3080258     }
3080259     //
3080260     //
3080261     //
3080262     if (header.data_offset == 0)
3080263     {
3080264         //
3080265         // Code and data segments are the same: no need
3080266         // to allocate more memory for the data segment.
3080267         //
3080268         allocated_data = 0;
3080269         process_domain_data = 0;
3080270     }
3080271     else
3080272     {
3080273         //
3080274         // Code and data segments are different: the
3080275         // data
3080276         // segment memory is allocated.
3080277         //
3080278         allocated_data = mb_alloc_size (process_domain_data);
3080279         if (allocated_data == 0)
3080280         {
3080281             //
3080282             // The separated program data (data segment)
3080283             // cannot be
3080284             // loaded into memory: free the already
3080285             // allocated memory
3080286             // for the program instructions, release the
3080287             // executable
3080288             // file inode and return with an error.
3080289             //
3080290             mb_free (allocated_text, process_domain_text);
3080291             if (DEBUG)
3080292             {
3080293                 k_printf ("%s:%i:mb_free(%i, %zi)",
```

```
3080294         __FILE__, __LINE__,
3080295         (unsigned int) allocated_text,
3080296         process_domain_data);
3080297     }
3080298     inode_put (inode);
3080299     errset (ENOMEM);      // Not enough space.
3080300     return (-1);
3080301 }
3080302 else if (DEBUG)
3080303     {
3080304         k_printf ("%s:%i:mb_alloc_size(%zi)",
3080305                 __FILE__, __LINE__,
3080306                 process_domain_data);
3080307     }
3080308 }
3080309 //
3080310 // Load executable in memory.
3080311 //
3080312 if (header.data_offset == 0)
3080313     {
3080314         //
3080315         // Code and data share the same segment.
3080316         //
3080317         for (eof = 0, memory_start = allocated_text,
3080318             off_inode = 0, size_read = 0;
3080319             off_inode < inode->size && !eof;
3080320             off_inode += size_read)
3080321             {
3080322                 memory_start += size_read;
3080323                 //
3080324                 // Read a block of memory.
3080325                 //
3080326                 size_read =
3080327                     inode_file_read (inode, off_inode, buffer,
3080328                                     MEM_BLOCK_SIZE, &eof);
3080329                 if (size_read < 0)
3080330                     {
```



```
3080331 //
3080332 // Free memory and inode.
3080333 //
3080334 mb_free (allocated_text, process_domain_text);
3080335 if (DEBUG)
3080336     {
3080337         k_printf ("%s:%i:mb_free(%i, %zi)",
3080338                 __FILE__, __LINE__,
3080339                 (unsigned int)
3080340                 allocated_text,
3080341                 process_domain_text);
3080342     }
3080343     inode_put (inode);
3080344     errset (EIO);
3080345     return (-1);
3080346 }
3080347 //
3080348 // Copy inside the right position to be
3080349 // executed.
3080350 //
3080351 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080352         memory_start, buffer,
3080353         (size_t) size_read, NULL);
3080354 }
3080355 }
3080356 else
3080357     {
3080358         //
3080359         // Code and data with different segments.
3080360         //
3080361         for (eof = 0, memory_start = allocated_text,
3080362             off_inode = 0, size_read = 0;
3080363             off_inode < process_domain_text && !eof;
3080364             off_inode += size_read)
3080365             {
3080366                 memory_start += size_read;
3080367                 //
```

```
3080368 // Read a block of memory
3080369 //
3080370 size_read =
3080371     inode_file_read (inode, off_inode, buffer,
3080372                     MEM_BLOCK_SIZE, &eof);
3080373 if (size_read < 0)
3080374     {
3080375         //
3080376         // Free memory and inode.
3080377         //
3080378         mb_free (allocated_text, process_domain_text);
3080379         if (DEBUG)
3080380             {
3080381                 k_printf ("%s:%i:mb_free(%i, %zi)",
3080382                           __FILE__, __LINE__,
3080383                           (unsigned int)
3080384                           allocated_text,
3080385                           process_domain_text);
3080386             }
3080387         mb_free (allocated_data, process_domain_data);
3080388         if (DEBUG)
3080389             {
3080390                 k_printf ("%s:%i:mb_free(%i, %zi)",
3080391                           __FILE__, __LINE__,
3080392                           (unsigned int)
3080393                           allocated_data,
3080394                           process_domain_data);
3080395             }
3080396         inode_put (inode);
3080397         errset (EIO);
3080398         return (-1);
3080399     }
3080400 //
3080401 // Copy inside the right position to be
3080402 // executed.
3080403 //
3080404 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
```

```
3080405         memory_start, buffer,
3080406         (size_t) size_read, NULL);
3080407     }
3080408     for (eof = 0, memory_start = allocated_data,
3080409         off_inode = header.data_offset, size_read =
3080410         0; off_inode < inode->size && !eof;
3080411         off_inode += size_read)
3080412     {
3080413         memory_start += size_read;
3080414         //
3080415         // Read a block of memory
3080416         //
3080417         size_read =
3080418             inode_file_read (inode, off_inode, buffer,
3080419                             MEM_BLOCK_SIZE, &eof);
3080420         if (size_read < 0)
3080421         {
3080422             //
3080423             // Free memory and inode.
3080424             //
3080425             mb_free (allocated_text, process_domain_text);
3080426             if (DEBUG)
3080427             {
3080428                 k_printf ("%s:%i:mb_free(%i, %zi)",
3080429                             __FILE__, __LINE__,
3080430                             (unsigned int)
3080431                             allocated_text,
3080432                             process_domain_text);
3080433             }
3080434             mb_free (allocated_data, process_domain_data);
3080435             if (DEBUG)
3080436             {
3080437                 k_printf ("%s:%i:mb_free(%i, %zi)",
3080438                             __FILE__, __LINE__,
3080439                             (unsigned int)
3080440                             allocated_data,
3080441                             process_domain_data);
```

```
3080442     }
3080443     inode_put (inode);
3080444     errset (EIO);
3080445     return (-1);
3080446     }
3080447     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080448           memory_start, buffer,
3080449           (size_t) size_read, NULL);
3080450     }
3080451     }
3080452     //
3080453     // The executable file was successfully loaded in
3080454     // memory:
3080455     // release the executable file inode.
3080456     //
3080457     inode_put (inode);
3080458     //
3080459     // Update process TEXT segment inside the GDT table.
3080460     //
3080461     gdt_segment (gdt_pid_to_segment_text (pid),
3080462                 (uint32_t) allocated_text,
3080463                 (uint32_t) (process_domain_text / 4096),
3080464                 1, 1, 0);
3080465     //
3080466     // Update process DATA segment inside the GDT table.
3080467     //
3080468     if (process_domain_data > 0)
3080469     {
3080470         gdt_segment (gdt_pid_to_segment_data (pid),
3080471                     (uint32_t) allocated_data,
3080472                     (uint32_t) (process_domain_data /
3080473                                 4096), 1, 0, 0);
3080474     }
3080475     else
3080476     {
3080477         gdt_segment (gdt_pid_to_segment_data (pid),
3080478                     (uint32_t) allocated_text,
```

```
3080479             (uint32_t) (process_domain_text /
3080480                     4096), 1, 0, 0);
3080481     }
3080482     //
3080483     // Calculate segment descriptors.
3080484     //
3080485     segment_text = (gdt_pid_to_segment_text (pid) << 3) + 0;
3080486     segment_data = (gdt_pid_to_segment_data (pid) << 3) + 0;
3080487     //
3080488     // Where is the stack?
3080489     //
3080490     if (process_domain_data > 0)
3080491     {
3080492         stack_location = allocated_data;
3080493     }
3080494     else
3080495     {
3080496         stack_location = allocated_text;
3080497     }
3080498     //
3080499     // Put environment data inside the stack.
3080500     //
3080501     new_sp -= env_data_size;          // -----
3080502     // ENVIRONMENT
3080503     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080504            (off_t) (stack_location + new_sp),
3080505            env_data, env_data_size, NULL);
3080506     //
3080507     // Put arguments data inside the stack.
3080508     //
3080509     new_sp -= arg_data_size;         // -----
3080510     // ARGUMENTS
3080511     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080512            (off_t) (stack_location + new_sp),
3080513            arg_data, arg_data_size, NULL);
3080514     //
3080515     // Put envp[] inside the stack, updating all the
```

```
3080516 // pointers.
3080517 //
3080518 new_sp -= 4; // ----- NULL
3080519 stack_element = (uint32_t) NULL;
3080520 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080521         (off_t) (stack_location + new_sp),
3080522         &stack_element, (sizeof stack_element), NULL);
3080523 //
3080524 // Calculate memory pointers from original relative
3080525 // pointers, inside the environment array of
3080526 // pointers.
3080527 //
3080528 p_off = new_sp;
3080529 p_off += 4;
3080530 p_off += arg_data_size;
3080531 for (i = 0; i < envc; i++)
3080532     {
3080533         envp[i] += p_off;
3080534     }
3080535 //
3080536 new_sp -= (envc * (sizeof (char *))); // ----- *envp[]
3080537 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080538         (off_t) (stack_location + new_sp),
3080539         envp, (envc * (sizeof (char *))), NULL);
3080540 //
3080541 // Save the envp[] location, needed in the
3080542 // following.
3080543 //
3080544 envp_address = new_sp;
3080545 //
3080546 // Put argv[] inside the stack, updating all the
3080547 // pointers.
3080548 //
3080549 new_sp -= 4; // ----- NULL
3080550 stack_element = (uint32_t) NULL;
3080551 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080552         (off_t) (stack_location + new_sp),
```

```
3080553         &stack_element, (sizeof stack_element), NULL);
3080554     //
3080555     // Calculate memory pointers from original relative
3080556     // pointers, inside the arguments array of pointers.
3080557     //
3080558     p_off = new_sp;
3080559     p_off += 4;
3080560     p_off += (envc * (sizeof (char *)));
3080561     p_off += 4;
3080562     for (i = 0; i < argc; i++)
3080563     {
3080564         argv[i] += p_off;
3080565     }
3080566     //
3080567     new_sp -= (argc * (sizeof (char *))); // ----- *argv[]
3080568     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080569             (off_t) (stack_location + new_sp),
3080570             argv, (argc * (sizeof (char *))), NULL);
3080571     //
3080572     // Save the argv[] location, needed in the
3080573     // following.
3080574     //
3080575     argv_address = new_sp;
3080576     //
3080577     // Put the pointer to the array envp[].
3080578     //
3080579     new_sp -= 4; // ----- argc
3080580     stack_element = envp_address;
3080581     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080582             (off_t) (stack_location + new_sp),
3080583             &stack_element, (sizeof stack_element), NULL);
3080584     //
3080585     // Put the pointer to the array argv[].
3080586     //
3080587     new_sp -= 4; // ----- argc
3080588     stack_element = argv_address;
3080589     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
```

```
3080590         (off_t) (stack_location + new_sp),
3080591         &stack_element, (sizeof stack_element), NULL);
3080592     //
3080593     // Put argc inside the stack.
3080594     //
3080595     new_sp -= 4; // ----- argc
3080596     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080597             (off_t) (stack_location + new_sp),
3080598             &argc, (sizeof argc), NULL);
3080599     //
3080600     // Set the rest of the stack.
3080601     //
3080602     new_sp -= 4; // ----- EFLAGS
3080603     stack_element = 0x0200;
3080604     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080605             (off_t) (stack_location + new_sp),
3080606             &stack_element, (sizeof stack_element), NULL);
3080607     new_sp -= 4; // ----- CS
3080608     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080609             (off_t) (stack_location + new_sp),
3080610             &segment_text, (sizeof segment_text), NULL);
3080611     new_sp -= 4; // ----- EIP
3080612     stack_element = 0;
3080613     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080614             (off_t) (stack_location + new_sp),
3080615             &stack_element, (sizeof stack_element), NULL);
3080616     new_sp -= 4; // ----- GS
3080617     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080618             (off_t) (stack_location + new_sp),
3080619             &segment_data, (sizeof segment_data), NULL);
3080620     new_sp -= 4; // ----- FS
3080621     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080622             (off_t) (stack_location + new_sp),
3080623             &segment_data, (sizeof segment_data), NULL);
3080624     new_sp -= 4; // ----- ES
3080625     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080626             (off_t) (stack_location + new_sp),
```



```
3080627         &segment_data, (sizeof segment_data), NULL);
3080628 new_sp -= 4; // ----- DS
3080629 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080630         (off_t) (stack_location + new_sp),
3080631         &segment_data, (sizeof segment_data), NULL);
3080632 new_sp -= 4; // ----- EDI
3080633 stack_element = 0;
3080634 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080635         (off_t) (stack_location + new_sp),
3080636         &stack_element, (sizeof stack_element), NULL);
3080637 new_sp -= 4; // ----- ESI
3080638 stack_element = 0;
3080639 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080640         (off_t) (stack_location + new_sp),
3080641         &stack_element, (sizeof stack_element), NULL);
3080642 new_sp -= 4; // ----- EBP
3080643 stack_element = 0;
3080644 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080645         (off_t) (stack_location + new_sp),
3080646         &stack_element, (sizeof stack_element), NULL);
3080647 new_sp -= 4; // ----- EBX
3080648 stack_element = 0;
3080649 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080650         (off_t) (stack_location + new_sp),
3080651         &stack_element, (sizeof stack_element), NULL);
3080652 new_sp -= 4; // ----- EDX
3080653 stack_element = 0;
3080654 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080655         (off_t) (stack_location + new_sp),
3080656         &stack_element, (sizeof stack_element), NULL);
3080657 new_sp -= 4; // ----- ECX
3080658 stack_element = 0;
3080659 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080660         (off_t) (stack_location + new_sp),
3080661         &stack_element, (sizeof stack_element), NULL);
3080662 new_sp -= 4; // ----- EAX
3080663 stack_element = 0;
```

```
3080664 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080665         (off_t) (stack_location + new_sp),
3080666         &stack_element, (sizeof stack_element), NULL);
3080667 //
3080668 // Close process file descriptors, if the
3080669 // 'FD_CLOEXEC' flag
3080670 // is present.
3080671 //
3080672 for (fdn = 0; fdn < OPEN_MAX; fdn++)
3080673 {
3080674     if (proc_table[pid].fd[0].file != NULL)
3080675     {
3080676         if (proc_table[pid].fd[0].fd_flags & FD_CLOEXEC)
3080677         {
3080678             s_close (pid, fdn);
3080679         }
3080680     }
3080681 }
3080682 //
3080683 // Select device for standard I/O, if a standard I/O
3080684 // stream must be
3080685 // opened.
3080686 //
3080687 if (proc_table[pid].device_tty != 0)
3080688 {
3080689     device = proc_table[pid].device_tty;
3080690 }
3080691 else
3080692 {
3080693     device = DEV_TTY;
3080694 }
3080695 //
3080696 // Prepare missing standard file descriptors. The
3080697 // function
3080698 // 'file_stdio_dev_make()' arranges the value for
3080699 // 'errno' if
3080700 // necessary. If a standard file descriptor cannot
```

```
3080701 // be allocated,
3080702 // the program is left without it.
3080703 //
3080704 if (proc_table[pid].fd[0].file == NULL)
3080705 {
3080706     file =
3080707         file_stdio_dev_make (device, S_IFCHR, O_RDONLY);
3080708     if (file != NULL) // stdin
3080709     {
3080710         proc_table[pid].fd[0].fl_flags = O_RDONLY;
3080711         proc_table[pid].fd[0].fd_flags = 0;
3080712         proc_table[pid].fd[0].file = file;
3080713         proc_table[pid].fd[0].file->offset = 0;
3080714     }
3080715 }
3080716 if (proc_table[pid].fd[1].file == NULL)
3080717 {
3080718     file =
3080719         file_stdio_dev_make (device, S_IFCHR, O_WRONLY);
3080720     if (file != NULL) // stdout
3080721     {
3080722         proc_table[pid].fd[1].fl_flags = O_WRONLY;
3080723         proc_table[pid].fd[1].fd_flags = 0;
3080724         proc_table[pid].fd[1].file = file;
3080725         proc_table[pid].fd[1].file->offset = 0;
3080726     }
3080727 }
3080728 if (proc_table[pid].fd[2].file == NULL)
3080729 {
3080730     file =
3080731         file_stdio_dev_make (device, S_IFCHR, O_WRONLY);
3080732     if (file != NULL) // stderr
3080733     {
3080734         proc_table[pid].fd[2].fl_flags = O_WRONLY;
3080735         proc_table[pid].fd[2].fd_flags = 0;
3080736         proc_table[pid].fd[2].file = file;
3080737         proc_table[pid].fd[2].file->offset = 0;
```

```
3080738     }
3080739 }
3080740 //
3080741 // Prepare to switch
3080742 //
3080743 previous_address_text = proc_table[pid].address_text;
3080744 previous_domain_text = proc_table[pid].domain_text;
3080745 previous_address_data = proc_table[pid].address_data;
3080746 previous_domain_data = proc_table[pid].domain_data;
3080747 previous_domain_stack = proc_table[pid].domain_stack;
3080748 previous_extra_data = proc_table[pid].extra_data;
3080749 //
3080750 proc_table[pid].address_text = allocated_text;
3080751 proc_table[pid].domain_text = process_domain_text;
3080752 proc_table[pid].address_data = allocated_data;
3080753 proc_table[pid].domain_data = process_domain_data;
3080754 proc_table[pid].domain_stack = process_domain_stack;
3080755 proc_table[pid].extra_data = (size_t) 0;
3080756 proc_table[pid].sp = new_sp;
3080757 strncpy (proc_table[pid].name, path, PATH_MAX);
3080758 //
3080759 // Ensure to have a terminated string.
3080760 //
3080761 proc_table[pid].name[PATH_MAX - 1] = 0;
3080762 //
3080763 // Reset 'sig_handler[]'.
3080764 //
3080765 for (sig = 0; sig < MAX_SIGNALS; sig++)
3080766     {
3080767         proc_table[pid].sig_handler[sig] = (uintptr_t) NULL;
3080768     }
3080769 //
3080770 // Free previous data memory (included stack).
3080771 //
3080772 if (previous_domain_data > 0)
3080773     {
3080774         mb_free (previous_address_data,
```

```
3080775         previous_domain_data + previous_extra_data);
3080776     if (DEBUG)
3080777     {
3080778         k_printf ("%s:%i:mb_free(%i, %zi)",
3080779                 __FILE__, __LINE__,
3080780                 (unsigned int)
3080781                 previous_address_data,
3080782                 previous_domain_data +
3080783                 previous_extra_data);
3080784     }
3080785 }
3080786 //
3080787 // Free code memory if not shared.
3080788 //
3080789 for (proc_count = 0, extra = 0; extra < PROCESS_MAX;
3080790     extra++)
3080791 {
3080792     if (proc_table[extra].status == PROC_EMPTY ||
3080793         proc_table[extra].status == PROC_ZOMBIE)
3080794     {
3080795         continue;
3080796     }
3080797     if (previous_address_text ==
3080798         proc_table[extra].address_text)
3080799     {
3080800         proc_count++;
3080801     }
3080802 }
3080803 if (proc_count == 0)
3080804 {
3080805     //
3080806     // The code segment can be released, because no
3080807     // other
3080808     // process is using it.
3080809     //
3080810     if (previous_domain_data > 0)
3080811     {
```

```
3080812     mb_free (previous_address_text,
3080813             previous_domain_text +
3080814             previous_extra_data);
3080815     if (DEBUG)
3080816     {
3080817         k_printf ("%s:%i:mb_free(%i, %zi)",
3080818                 __FILE__, __LINE__,
3080819                 (unsigned int)
3080820                 previous_address_text,
3080821                 previous_domain_text +
3080822                 previous_extra_data);
3080823     }
3080824 }
3080825 else
3080826 {
3080827     mb_free (previous_address_text,
3080828             previous_domain_text);
3080829     if (DEBUG)
3080830     {
3080831         k_printf ("%s:%i:mb_free(%i, %zi)",
3080832                 __FILE__, __LINE__,
3080833                 (unsigned int)
3080834                 previous_address_text,
3080835                 previous_domain_text);
3080836     }
3080837 }
3080838 }
3080839 //
3080840 // Change the segment and the stack pointer, from
3080841 // the interrupt.
3080842 // [1] Anyway, the stack segment selector does not
3080843 // change.
3080844 //
3080845 proc_stack_segment_selector = segment_data;    // [1]
3080846 proc_stack_pointer = proc_table[pid].sp;
3080847 //
3080848 //
```

```
3080849 //
3080850 return (0);
3080851 }
```

94.14.23 kernel/proc/proc_timer_init.c

Si veda la sezione [93.20.22](#).

```
3090001 #include <kernel/proc.h>
3090002 #include <stdint.h>
3090003 #include <kernel/lib_k.h>
3090004 #include <kernel/ibm_i386.h>
3090005 #include <stdint.h>
3090006 //-----
3090007 void
3090008 proc_timer_init (clock_t freq)
3090009 {
3090010     int input_freq = 1193180;
3090011     //
3090012     // La frequenza di riferimento è 1,19318 MHz, la
3090013     // quale va
3090014     // divisa per la frequenza che si intende avere
3090015     // effettivamente.
3090016     //
3090017     int divisor = input_freq / freq;
3090018     //
3090019     // Il risultato deve essere un valore intero
3090020     // maggiore di zero
3090021     // e inferiore di UINT16_MAX, altrimenti è stata
3090022     // chiesta una
3090023     // frequenza troppo elevata o troppo bassa.
3090024     //
3090025     if (divisor == 0 || divisor > UINT16_MAX)
3090026     {
3090027         k_printf
3090028             ("%s] ERROR: IRQ 0 frequency wrong: %i Hz!\n"
3090029             "%s]           The min allowed frequency \n"
```

```

3090030         "[%s]          is 18.22 Hz.\n",
3090031         "[%s]          The max allowed frequency \n"
3090032         "[%s]          is 1.19 MHz.\n",
3090033         __func__, freq, __func__, __func__, __func__,
3090034         __func__);
3090035     return;
3090036 }
3090037 //
3090038 // Il valore che si ottiene, ovvero il «divisore»,
3090039 // va
3090040 // comunicato al PIT (programmable interval timer),
3090041 // spezzandolo in due parti.
3090042 //
3090043 out_8 ((uint32_t) 0x43, (uint32_t) 0x36);
3090044 //
3090045 // Lower byte.
3090046 //
3090047 out_8 ((uint32_t) 0x40, (uint32_t) (divisor & 0x0F));
3090048 //
3090049 // Higher byte.
3090050 //
3090051 out_8 ((uint32_t) 0x40, (uint32_t) (divisor / 0x10));
3090052 }

```

94.14.24 kernel/proc/proc_wakeup_pipe_read.c

«

Si veda la sezione [93.20.23](#).

```

3100001 #include <kernel/proc.h>
3100002 //-----
3100003 void
3100004 proc_wakeup_pipe_read (inode_t * inode)
3100005 {
3100006     pid_t pid;
3100007
3100008     for (pid = 1; pid < PROCESS_MAX; pid++)
3100009     {

```



```
3100010     if ((proc_table[pid].status == PROC_SLEEPING)
3100011         && (proc_table[pid].wakeup_events
3100012           & WAKEUP_EVENT_PIPE_READ)
3100013         && (proc_table[pid].wakeup_inode == inode))
3100014     {
3100015         proc_table[pid].wakeup_events = 0;
3100016         proc_table[pid].wakeup_inode = NULL;
3100017         proc_table[pid].status = PROC_READY;
3100018     }
3100019 }
3100020 }
```

94.14.25 kernel/proc/proc_wakeup_pipe_write.c



Si veda la sezione [93.20.23](#).

```
3110001 #include <kernel/proc.h>
3110002 //-----
3110003 void
3110004 proc_wakeup_pipe_write (inode_t * inode)
3110005 {
3110006     pid_t pid;
3110007
3110008     for (pid = 1; pid < PROCESS_MAX; pid++)
3110009     {
3110010         if ((proc_table[pid].status == PROC_SLEEPING)
3110011             && (proc_table[pid].wakeup_events
3110012               & WAKEUP_EVENT_PIPE_WRITE)
3110013             && (proc_table[pid].wakeup_inode == inode))
3110014         {
3110015             proc_table[pid].wakeup_events = 0;
3110016             proc_table[pid].wakeup_inode = NULL;
3110017             proc_table[pid].status = PROC_READY;
3110018         }
3110019     }
3110020 }
```

94.14.26 kernel/proc/proc_wakeup_terminal.c

<<

Si veda la sezione [93.20.23](#).

```
3120001 #include <kernel/proc.h>
3120002 #include <kernel/lib_k.h>
3120003 #include <sys/types.h>
3120004 //-----
3120005 void
3120006 proc_wakeup_terminal (void)
3120007 {
3120008     pid_t pid;
3120009     int maj;
3120010     //
3120011     // At the moment, all processes waiting for reading
3120012     // a terminal
3120013     // or the console are reactivated.
3120014     //
3120015     for (pid = 0; pid < PROCESS_MAX; pid++)
3120016     {
3120017         if ((proc_table[pid].status == PROC_SLEEPING)
3120018             && (proc_table[pid].wakeup_events &
3120019                 WAKEUP_EVENT_DEV_READ))
3120020         {
3120021             maj = major (proc_table[pid].wakeup_dev);
3120022             if (maj == DEV_TTY_MAJOR
3120023                 || maj == DEV_CONSOLE_MAJOR)
3120024             {
3120025                 //
3120026                 // A process waiting for that terminal
3120027                 // was found:
3120028                 // remove the waiting event and set it
3120029                 // ready.
3120030                 //
3120031                 proc_table[pid].wakeup_events &=
3120032                     ~WAKEUP_EVENT_DEV_READ;
3120033                 proc_table[pid].wakeup_dev = 0;
3120034                 proc_table[pid].status = PROC_READY;
```

```
3120035     }
3120036     }
3120037 }
3120038 }
```

94.14.27 kernel/proc/ptr.c



Si veda la sezione [93.20.27](#).

```
3130001 #include <kernel/proc.h>
3130002 #include <kernel/lib_s.h>
3130003 #include <kernel/lib_k.h>
3130004 #include <stdint.h>
3130005 //-----
3130006 #define DEBUG 0
3130007 //-----
3130008 void *
3130009 ptr (pid_t pid, void *p)
3130010 {
3130011     uintptr_t start;
3130012     //
3130013     if (p == NULL)
3130014     {
3130015         return (NULL);
3130016     }
3130017     else if (proc_table[pid].domain_data == 0)
3130018     {
3130019         start = proc_table[pid].address_text;
3130020     }
3130021     else
3130022     {
3130023         start = proc_table[pid].address_data;
3130024     }
3130025     //
3130026     return ((void *) (start + (uintptr_t) p));
3130027 }
```

94.14.28 kernel/proc/sysroutine.c



Si veda la sezione [93.20.28](#).

```
3140001 #include <kernel/proc.h>
3140002 #include <errno.h>
3140003 #include <kernel/lib_k.h>
3140004 #include <kernel/lib_s.h>
3140005 #include <stdint.h>
3140006 //-----
3140007 static void sysroutine_error_back (int *number,
3140008                                   int *line,
3140009                                   char *file_name);
3140010 //-----
3140011 void
3140012 sysroutine (uint32_t syscallnr, uint32_t msg_off,
3140013            uint32_t msg_size)
3140014 {
3140015     pid_t pid = proc_current;
3140016     //
3140017     // Inbox.
3140018     //
3140019     union
3140020     {
3140021         sysmsg_accept_t accept;
3140022         sysmsg_bind_t bind;
3140023         sysmsg_brk_t brk;
3140024         sysmsg_chdir_t chdir;
3140025         sysmsg_chmod_t chmod;
3140026         sysmsg_chown_t chown;
3140027         sysmsg_clock_t clock;
3140028         sysmsg_close_t close;
3140029         sysmsg_connect_t connect;
3140030         sysmsg_dup_t dup;
3140031         sysmsg_dup2_t dup2;
3140032         sysmsg_exec_t exec;
3140033         sysmsg_exit_t exit;
3140034         sysmsg_fchmod_t fchmod;
```

```
3140035 sysmsg_fchown_t fchown;
3140036 sysmsg_fcntl_t fcntl;
3140037 sysmsg_fork_t fork;
3140038 sysmsg_fstat_t fstat;
3140039 sysmsg_ipconfig_t ipconfig;
3140040 sysmsg_jump_t jmp;
3140041 sysmsg_kill_t kill;
3140042 sysmsg_link_t link;
3140043 sysmsg_listen_t listen;
3140044 sysmsg_lseek_t lseek;
3140045 sysmsg_mkdir_t mkdir;
3140046 sysmsg_mknod_t mknod;
3140047 sysmsg_mount_t mount;
3140048 sysmsg_open_t open;
3140049 sysmsg_pipe_t pipe;
3140050 sysmsg_read_t read;
3140051 sysmsg_recvfrom_t recvfrom;
3140052 sysmsg_route_t route;
3140053 sysmsg_send_t send;
3140054 sysmsg_sbrk_t sbrk;
3140055 sysmsg_seteuid_t seteuid;
3140056 sysmsg_setuid_t setuid;
3140057 sysmsg_setegid_t setegid;
3140058 sysmsg_setgid_t setgid;
3140059 sysmsg_signal_t signal;
3140060 sysmsg_sleep_t sleep;
3140061 sysmsg_socket_t socket;
3140062 sysmsg_stat_t stat;
3140063 sysmsg_stime_t stime;
3140064 sysmsg_tcatrr_t tcatrr;
3140065 sysmsg_time_t time;
3140066 sysmsg_uarea_t uarea;
3140067 sysmsg_umask_t umask;
3140068 sysmsg_umount_t umount;
3140069 sysmsg_unlink_t unlink;
3140070 sysmsg_wait_t wait;
3140071 sysmsg_write_t write;
```

```
3140072     sysmsg_zpchar_t zpchar;
3140073     sysmsg_zpstring_t zpstring;
3140074 } *msg;
3140075 //
3140076 // Align the message address pointer to the source
3140077 // message.
3140078 //
3140079 msg = ptr (pid, (void *) msg_off);
3140080 //
3140081 // Verify if the system call was emitted from kernel
3140082 // code.
3140083 // The kernel can emit only some particular system
3140084 // call:
3140085 // SYS_NULL, to let other processes run;
3140086 // SYS_FORK, to let fork itself;
3140087 // SYS_EXEC, to replace a forked copy of itself.
3140088 //
3140089 if (pid == 0)
3140090     {
3140091         //
3140092         // This is the kernel code!
3140093         //
3140094         if (syscallnr != SYS_0
3140095             && syscallnr != SYS_FORK
3140096             && syscallnr != SYS_EXEC
3140097             && syscallnr != SYS_ZPSTRING)
3140098             {
3140099                 k_printf
3140100                 ("kernel panic: the system call %i ",
3140101                 syscallnr);
3140102                 k_printf
3140103                 ("was received while running "
3140104                 "in kernel space!\n");
3140105             }
3140106     }
3140107 //
3140108 // Entering a system call, the kernel variable
```

```
3140109 // 'errno' must be
3140110 // reset, otherwise, a previous kernel code error
3140111 // might be returned
3140112 // to the applications.
3140113 //
3140114 errno = 0;
3140115 errln = 0;
3140116 errfn[0] = 0;
3140117 //
3140118 // Do the request from the received system call.
3140119 //
3140120 switch (syscallnr)
3140121 {
3140122 case SYS_0:
3140123     break;
3140124 case SYS_ACCEPT:
3140125     msg->accept.ret =
3140126         s_accept (pid, msg->accept.sfdn,
3140127                 &msg->accept.addr, &msg->accept.addrlen);
3140128     msg->accept.fl_flags =
3140129         proc_table[pid].fd[msg->accept.sfdn].fl_flags;
3140130     sysroutine_error_back (&msg->accept.errno,
3140131                          &msg->accept.errln,
3140132                          msg->accept.errfn);
3140133     break;
3140134 case SYS_BIND:
3140135     msg->bind.ret = s_bind (pid, msg->bind.sfdn,
3140136                          &msg->bind.addr,
3140137                          msg->bind.addrlen);
3140138     sysroutine_error_back (&msg->bind.errno,
3140139                          &msg->bind.errln,
3140140                          msg->bind.errfn);
3140141     break;
3140142 case SYS_BRK:
3140143     msg->brk.ret = s_brk (pid, msg->brk.address);
3140144     sysroutine_error_back (&msg->brk.errno,
3140145                          &msg->brk.errln,
```

```
3140146             msg->brk.errfn);
3140147     break;
3140148 case SYS_CHDIR:
3140149     msg->chdir.ret =
3140150         s_chdir (pid, ptr (pid, (void *) msg->chdir.path));
3140151     sysroutine_error_back (&msg->chdir.errno,
3140152                             &msg->chdir.errln,
3140153                             msg->chdir.errfn);
3140154     break;
3140155 case SYS_CHMOD:
3140156     msg->chmod.ret = s_chmod (pid,
3140157                             ptr (pid,
3140158                                 (void *) msg->
3140159                                     chmod.path),
3140160                             msg->chmod.mode);
3140161     sysroutine_error_back (&msg->chmod.errno,
3140162                             &msg->chmod.errln,
3140163                             msg->chmod.errfn);
3140164     break;
3140165 case SYS_CHOWN:
3140166     msg->chown.ret = s_chown (pid,
3140167                             ptr (pid,
3140168                                 (void *) msg->
3140169                                     chown.path),
3140170                             msg->chown.uid,
3140171                             msg->chown.gid);
3140172     sysroutine_error_back (&msg->chown.errno,
3140173                             &msg->chown.errln,
3140174                             msg->chown.errfn);
3140175     break;
3140176 case SYS_CLOCK:
3140177     msg->clock.ret = s_clock (pid);
3140178     break;
3140179 case SYS_CLOSE:
3140180     msg->close.ret = s_close (pid, msg->close.fdn);
3140181     sysroutine_error_back (&msg->close.errno,
3140182                             &msg->close.errln,
```



```
3140183         msg->close.errfn);
3140184     break;
3140185     case SYS_CONNECT:
3140186         msg->connect.ret =
3140187             s_connect (pid, msg->connect.sfdn,
3140188                 &msg->connect.addr,
3140189                 msg->connect.addrlen);
3140190         sysroutine_error_back (&msg->connect.errno,
3140191             &msg->connect.errln,
3140192             msg->connect.errfn);
3140193     break;
3140194     case SYS_DUP:
3140195         msg->dup.ret = s_dup (pid, msg->dup.fdn_old);
3140196         sysroutine_error_back (&msg->dup.errno,
3140197             &msg->dup.errln,
3140198             msg->dup.errfn);
3140199     break;
3140200     case SYS_DUP2:
3140201         msg->dup2.ret = s_dup2 (pid, msg->dup2.fdn_old,
3140202             msg->dup2.fdn_new);
3140203         sysroutine_error_back (&msg->dup2.errno,
3140204             &msg->dup2.errln,
3140205             msg->dup2.errfn);
3140206     break;
3140207     case SYS_EXEC:
3140208         msg->exec.ret = proc_sys_exec (pid,
3140209             ptr (pid,
3140210                 (void *)
3140211                 msg->exec.path),
3140212             msg->exec.argc,
3140213             msg->exec.arg_data,
3140214             msg->exec.envc,
3140215             msg->exec.env_data);
3140216         msg->exec.uid = proc_table[pid].uid;
3140217         msg->exec.euid = proc_table[pid].euid;
3140218         sysroutine_error_back (&msg->exec.errno,
3140219             &msg->exec.errln,
```

```
3140220                                 msg->exec.errfn);
3140221         break;
3140222     case SYS_EXIT:
3140223         if (pid == 0)
3140224             {
3140225                 k_printf ("kernel alert: "
3140226                           "the kernel cannot exit!\n");
3140227             }
3140228         else
3140229             {
3140230                 s__exit (pid, msg->exit.status);
3140231             }
3140232         break;
3140233     case SYS_FCHMOD:
3140234         msg->fchmod.ret = s_fchmod (pid, msg->fchmod.fdn,
3140235                                     msg->fchmod.mode);
3140236         sysroutine_error_back (&msg->fchmod.errno,
3140237                                 &msg->fchmod.errln,
3140238                                 msg->fchmod.errfn);
3140239         break;
3140240     case SYS_FCHOWN:
3140241         msg->fchown.ret = s_fchown (pid, msg->fchown.fdn,
3140242                                     msg->fchown.uid,
3140243                                     msg->fchown.gid);
3140244         sysroutine_error_back (&msg->fchown.errno,
3140245                                 &msg->fchown.errln,
3140246                                 msg->fchown.errfn);
3140247         break;
3140248     case SYS_FCNTL:
3140249         msg->fcntl.ret = s_fcntl (pid, msg->fcntl.fdn,
3140250                                     msg->fcntl.cmd,
3140251                                     msg->fcntl.arg);
3140252         sysroutine_error_back (&msg->fcntl.errno,
3140253                                 &msg->fcntl.errln,
3140254                                 msg->fcntl.errfn);
3140255         break;
3140256     case SYS_FORK:
```

```
3140257     msg->fork.ret = s_fork (pid);
3140258     sysroutine_error_back (&msg->fork.errno,
3140259                          &msg->fork.errln,
3140260                          msg->fork.errfn);
3140261     break;
3140262 case SYS_FSTAT:
3140263     msg->fstat.ret =
3140264         s_fstat (pid, msg->fstat.fdn, &msg->fstat.stat);
3140265     sysroutine_error_back (&msg->fstat.errno,
3140266                          &msg->fstat.errln,
3140267                          msg->fstat.errfn);
3140268     break;
3140269 case SYS_IPCONFIG:
3140270     msg->ipconfig.ret =
3140271         s_ipconfig (pid, msg->ipconfig.n,
3140272                  msg->ipconfig.address, msg->ipconfig.m);
3140273     sysroutine_error_back (&msg->ipconfig.errno,
3140274                          &msg->ipconfig.errln,
3140275                          msg->ipconfig.errfn);
3140276     break;
3140277 case SYS_KILL:
3140278     msg->kill.ret =
3140279         s_kill (pid, msg->kill.pid, msg->kill.signal);
3140280     sysroutine_error_back (&msg->kill.errno,
3140281                          &msg->kill.errln,
3140282                          msg->kill.errfn);
3140283     break;
3140284 case SYS_LINK:
3140285     msg->link.ret
3140286         = s_link (pid,
3140287                 ptr (pid,
3140288                     (void *) msg->link.path_old),
3140289                 ptr (pid, (void *) msg->link.path_new));
3140290     sysroutine_error_back (&msg->link.errno,
3140291                          &msg->link.errln,
3140292                          msg->link.errfn);
3140293     break;
```

```
3140294     case SYS_LISTEN:
3140295         msg->listen.ret =
3140296             s_listen (pid, msg->listen.sfdn,
3140297                     msg->listen.backlog);
3140298         sysroutine_error_back (&msg->listen.errno,
3140299                               &msg->listen.errln,
3140300                               msg->listen.errfn);
3140301         break;
3140302     case SYS_LONGJMP:
3140303         s_longjmp (pid, msg->jmp.env, msg->jmp.ret);
3140304         break;
3140305     case SYS_LSEEK:
3140306         msg->lseek.ret = s_lseek (pid, msg->lseek.fdn,
3140307                                  msg->lseek.offset,
3140308                                  msg->lseek.whence);
3140309         sysroutine_error_back (&msg->lseek.errno,
3140310                               &msg->lseek.errln,
3140311                               msg->lseek.errfn);
3140312         break;
3140313     case SYS_MKDIR:
3140314         msg->mkdir.ret = s_mkdir (pid,
3140315                                   ptr (pid,
3140316                                         (void *) msg->
3140317                                             mkdir.path),
3140318                                   msg->mkdir.mode);
3140319         sysroutine_error_back (&msg->mkdir.errno,
3140320                               &msg->mkdir.errln,
3140321                               msg->mkdir.errfn);
3140322         break;
3140323     case SYS_MKNOD:
3140324         msg->mknod.ret = s_mknod (pid,
3140325                                   ptr (pid,
3140326                                         (void *) msg->
3140327                                             mknod.path),
3140328                                   msg->mknod.mode,
3140329                                   msg->mknod.device);
3140330         sysroutine_error_back (&msg->mknod.errno,
```

```
3140331         &msg->mknod.errln,
3140332         msg->mknod.errfn);
3140333     break;
3140334 case SYS_MOUNT:
3140335     msg->mount.ret = s_mount (pid,
3140336                             ptr (pid,
3140337                                 (void *) msg->
3140338                                 mount.path_dev),
3140339                             ptr (pid,
3140340                                 (void *) msg->
3140341                                 mount.path_mnt),
3140342                             msg->mount.options);
3140343     sysroutine_error_back (&msg->mount.errno,
3140344                           &msg->mount.errln,
3140345                           msg->mount.errfn);
3140346     break;
3140347 case SYS_OPEN:
3140348     msg->open.ret = s_open (pid,
3140349                            ptr (pid,
3140350                                (void *) msg->open.path),
3140351                            msg->open.flags,
3140352                            msg->open.mode);
3140353     sysroutine_error_back (&msg->open.errno,
3140354                           &msg->open.errln,
3140355                           msg->open.errfn);
3140356     break;
3140357 case SYS_PIPE:
3140358     msg->pipe.ret = s_pipe (pid, msg->pipe.pipefd);
3140359     sysroutine_error_back (&msg->pipe.errno,
3140360                           &msg->pipe.errln,
3140361                           msg->pipe.errfn);
3140362     break;
3140363 case SYS_PGRP:
3140364     proc_table[pid].pgrp = pid;
3140365     break;
3140366 case SYS_READ:
3140367     msg->read.ret = s_read (pid, msg->read.fdn,
```

```
3140368         ptr (pid,
3140369             msg->read.buffer),
3140370         msg->read.count);
3140371     msg->read.fl_flags =
3140372         proc_table[pid].fd[msg->read.fdn].fl_flags;
3140373     sysroutine_error_back (&msg->read.errno,
3140374                             &msg->read.errln,
3140375                             msg->read.errfn);
3140376     break;
3140377 case SYS_RECVFROM:
3140378     msg->recvfrom.ret =
3140379         s_recvfrom
3140380         (pid, msg->recvfrom.sfdn,
3140381         ptr (pid, msg->recvfrom.buffer),
3140382         msg->recvfrom.count,
3140383         msg->recvfrom.flags, ptr (pid,
3140384                                 msg->recvfrom.addrfrom),
3140385         ptr (pid, msg->recvfrom.addrsz));
3140386     msg->recvfrom.fl_flags =
3140387         proc_table[pid].fd[msg->recvfrom.sfdn].fl_flags;
3140388     sysroutine_error_back (&msg->recvfrom.errno,
3140389                             &msg->recvfrom.errln,
3140390                             msg->recvfrom.errfn);
3140391     break;
3140392 case SYS_ROUTEADD:
3140393     msg->route.ret =
3140394         s_routeadd (pid, msg->route.destination,
3140395                     msg->route.m, msg->route.router,
3140396                     msg->route.device);
3140397     sysroutine_error_back (&msg->route.errno,
3140398                             &msg->route.errln,
3140399                             msg->route.errfn);
3140400     break;
3140401 case SYS_ROUTEDEL:
3140402     msg->route.ret =
3140403         s_routedel (pid, msg->route.destination,
3140404                     msg->route.m);
```

```
3140405     sysroutine_error_back (&msg->route.errno,
3140406                             &msg->route.errln,
3140407                             msg->route.errfn);
3140408     break;
3140409 case SYS_SBRK:
3140410     msg->sbrk.ret = s_sbrk (pid, msg->sbrk.increment);
3140411     sysroutine_error_back (&msg->sbrk.errno,
3140412                             &msg->sbrk.errln,
3140413                             msg->sbrk.errfn);
3140414     break;
3140415 case SYS_SEND:
3140416     msg->send.ret = s_send (pid, msg->send.sfdn,
3140417                             ptr (pid,
3140418                                 (void *) msg->send.
3140419                                 buffer), msg->send.count,
3140420                                 msg->send.flags);
3140421     sysroutine_error_back (&msg->send.errno,
3140422                             &msg->send.errln,
3140423                             msg->send.errfn);
3140424     break;
3140425 case SYS_SETEUID:
3140426     msg->seteuid.ret = s_seteuid (pid, msg->seteuid.euid);
3140427     msg->seteuid.euid = proc_table[pid].euid;
3140428     sysroutine_error_back (&msg->seteuid.errno,
3140429                             &msg->seteuid.errln,
3140430                             msg->seteuid.errfn);
3140431     break;
3140432 case SYS_SETUID:
3140433     msg->setuid.ret = s_setuid (pid, msg->setuid.euid);
3140434     msg->setuid.uid = proc_table[pid].uid;
3140435     msg->setuid.euid = proc_table[pid].euid;
3140436     msg->setuid.suid = proc_table[pid].suid;
3140437     sysroutine_error_back (&msg->setuid.errno,
3140438                             &msg->setuid.errln,
3140439                             msg->setuid.errfn);
3140440     break;
3140441 case SYS_SETEGID:
```

```
3140442     msg->setegid.ret = s_setegid (pid, msg->setegid.egid);
3140443     msg->setegid.egid = proc_table[pid].egid;
3140444     sysroutine_error_back (&msg->setegid.errno,
3140445                             &msg->setegid.errln,
3140446                             msg->setegid.errfn);
3140447     break;
3140448 case SYS_SETGID:
3140449     msg->setgid.ret = s_setgid (pid, msg->setgid.egid);
3140450     msg->setgid.gid = proc_table[pid].gid;
3140451     msg->setgid.egid = proc_table[pid].egid;
3140452     msg->setgid.sgid = proc_table[pid].sgid;
3140453     sysroutine_error_back (&msg->setgid.errno,
3140454                             &msg->setgid.errln,
3140455                             msg->setgid.errfn);
3140456     break;
3140457 case SYS_SETJMP:
3140458     msg->jmp.ret = s_setjmp (pid, msg->jmp.env);
3140459     break;
3140460 case SYS_SIGNAL:
3140461     msg->signal.ret =
3140462         s_signal (pid, msg->signal.signal,
3140463                 msg->signal.handler, msg->signal.wrapper);
3140464     sysroutine_error_back (&msg->signal.errno,
3140465                             &msg->signal.errln,
3140466                             msg->signal.errfn);
3140467     break;
3140468 case SYS_SLEEP:
3140469     proc_table[pid].status = PROC_SLEEPING;
3140470     proc_table[pid].ret = 0;
3140471     proc_table[pid].wakeup_events = msg->sleep.events;
3140472     proc_table[pid].wakeup_timer = msg->sleep.seconds;
3140473     break;
3140474 case SYS_STAT:
3140475     msg->stat.ret = s_stat (pid,
3140476                             ptr (pid,
3140477                                 (void *) msg->stat.path),
3140478                             &msg->stat.stat);
```



```
3140479     sysroutine_error_back (&msg->stat.errno,  
3140480                             &msg->stat.errln,  
3140481                             msg->stat.errfn);  
3140482     break;  
3140483     case SYS_STIME:  
3140484         msg->stime.ret = s_stime (pid, &msg->stime.timer);  
3140485         break;  
3140486     case SYS_TCGETATTR:  
3140487         msg->tcattr.ret =  
3140488             s_tcgetattr (pid, msg->tcattr.fdn,  
3140489                             ptr (pid, msg->tcattr.attr));  
3140490         sysroutine_error_back (&msg->tcattr.errno,  
3140491                             &msg->tcattr.errln,  
3140492                             msg->tcattr.errfn);  
3140493         break;  
3140494     case SYS_TCSETATTR:  
3140495         msg->tcattr.ret =  
3140496             s_tcsetattr (pid, msg->tcattr.fdn,  
3140497                             msg->tcattr.action, ptr (pid,  
3140498                                                         msg->tcattr.  
3140499                                                         attr));  
3140500         sysroutine_error_back (&msg->tcattr.errno,  
3140501                             &msg->tcattr.errln,  
3140502                             msg->tcattr.errfn);  
3140503         break;  
3140504     case SYS_TIME:  
3140505         msg->time.ret = s_time (pid, NULL);  
3140506         break;  
3140507     case SYS_UAREA:  
3140508         msg->uarea.uid = proc_table[pid].uid;  
3140509         msg->uarea.suid = proc_table[pid].suid;  
3140510         msg->uarea.euid = proc_table[pid].euid;  
3140511         msg->uarea.gid = proc_table[pid].gid;  
3140512         msg->uarea.sgid = proc_table[pid].sgid;  
3140513         msg->uarea.egid = proc_table[pid].egid;  
3140514         msg->uarea.pid = pid;  
3140515         msg->uarea.ppid = proc_table[pid].ppid;
```

```
3140516     msg->uarea.pgrp = proc_table[pid].pgrp;
3140517     msg->uarea.umask = proc_table[pid].umask;
3140518     strncpy (ptr (pid, msg->uarea.path_cwd),
3140519             proc_table[pid].path_cwd,
3140520             msg->uarea.path_cwd_size);
3140521     break;
3140522 case SYS_UMASK:
3140523     msg->umask.ret = proc_table[pid].umask;
3140524     proc_table[pid].umask = (msg->umask.umask & 00777);
3140525     break;
3140526 case SYS_UMOUNT:
3140527     msg->umount.ret = s_umount (pid,
3140528                                ptr (pid,
3140529                                    (void *)
3140530                                    msg->umount.
3140531                                    path_mnt));
3140532     sysroutine_error_back (&msg->umount.errno,
3140533                            &msg->umount.errln,
3140534                            msg->umount.errfn);
3140535     break;
3140536 case SYS_UNLINK:
3140537     msg->unlink.ret = s_unlink (pid,
3140538                                ptr (pid,
3140539                                    (void *) msg->
3140540                                    unlink.path));
3140541     sysroutine_error_back (&msg->unlink.errno,
3140542                            &msg->unlink.errln,
3140543                            msg->unlink.errfn);
3140544     break;
3140545 case SYS_WAIT:
3140546     msg->wait.ret = s_wait (pid, &msg->wait.status);
3140547     sysroutine_error_back (&msg->wait.errno,
3140548                            &msg->wait.errln,
3140549                            msg->wait.errfn);
3140550     break;
3140551 case SYS_WRITE:
3140552     msg->write.ret = s_write (pid, msg->write.fdn,
```

```
3140553         ptr (pid,
3140554             (void *) msg->
3140555             write.buffer),
3140556             msg->write.count);
3140557     sysroutine_error_back (&msg->write.errno,
3140558                           &msg->write.errln,
3140559                           msg->write.errfn);
3140560     break;
3140561 case SYS_SOCKET:
3140562     msg->socket.ret =
3140563         s_socket (pid, msg->socket.family,
3140564                 msg->socket.type, msg->socket.protocol);
3140565     sysroutine_error_back (&msg->socket.errno,
3140566                           &msg->socket.errln,
3140567                           msg->socket.errfn);
3140568     break;
3140569 case SYS_ZPCHAR:
3140570     dev_io (pid, DEV_TTY, DEV_WRITE, 0L,
3140571           &msg->zpchar.c, 1, NULL);
3140572     break;
3140573 case SYS_ZPSTRING:
3140574     dev_io (pid, DEV_TTY, DEV_WRITE, (off_t) 0,
3140575           msg->zpstring.string,
3140576           strlen (msg->zpstring.string), NULL);
3140577     break;
3140578 default:
3140579     k_printf
3140580         ("kernel alert: unknown system call %i!\n",
3140581         syscallnr);
3140582     break;
3140583 }
3140584 //
3140585 // Continue with the scheduler.
3140586 //
3140587 proc_scheduler ();
3140588 }
3140589
```

```
3140590 //-----  
3140591 static void  
3140592 sysroutine_error_back (int *number, int *line,  
3140593                        char *file_name)  
3140594 {  
3140595     *number = errno;  
3140596     *line = errln;  
3140597     strncpy (file_name, errfn, PATH_MAX);  
3140598     file_name[PATH_MAX - 1] = 0;  
3140599 }
```

Sorgenti della libreria generale



95.1	os32: file isolati della directory «lib/»	1799
95.1.1	lib/NULL.h	1799
95.1.2	lib/SEEK.h	1800
95.1.3	lib/assert.h	1800
95.1.4	lib/clock_t.h	1801
95.1.5	lib/ctype.h	1801
95.1.6	lib/limits.h	1802
95.1.7	lib/ptrdiff_t.h	1804
95.1.8	lib/restrict.h	1804
95.1.9	lib/size_t.h	1805
95.1.10	lib/stdarg.h	1805
95.1.11	lib/stdbool.h	1806
95.1.12	lib/stddef.h	1806
95.1.13	lib/stdint.h	1807
95.1.14	lib/time_t.h	1810
95.1.15	lib/wchar_t.h	1810
95.2	os32: «lib/_gcc.h»	1811
95.2.1	lib/_gcc/__divdi3.c	1812
95.2.2	lib/_gcc/__moddi3.c	1812
95.2.3	lib/_gcc/__udivdi3.c	1813
95.2.4	lib/_gcc/__umoddi3.c	1813
95.2.5	lib/_gcc/_lldiv.c	1813

95.2.6	lib/_gcc/_ulldiv.c	1815
95.3	os32: «lib/arpa/inet.h»	1818
95.3.1	lib/arpa/inet/htonl.c	1819
95.3.2	lib/arpa/inet/htons.c	1819
95.3.3	lib/arpa/inet/inet_ntop.c	1820
95.3.4	lib/arpa/inet/inet_pton.c	1821
95.3.5	lib/arpa/inet/ntohl.c	1824
95.3.6	lib/arpa/inet/ntohs.c	1825
95.4	os32: «lib/dirent.h»	1825
95.4.1	lib/dirent/DIR.c	1827
95.4.2	lib/dirent/closedir.c	1828
95.4.3	lib/dirent/opendir.c	1829
95.4.4	lib/dirent/readdir.c	1832
95.4.5	lib/dirent/rewinddir.c	1834
95.5	os32: «lib/errno.h»	1835
95.5.1	lib/errno/errno.c	1846
95.6	os32: «lib/fcntl.h»	1846
95.6.1	lib/fcntl/creat.c	1850
95.6.2	lib/fcntl/fcntl.c	1850
95.6.3	lib/fcntl/open.c	1852
95.7	os32: «lib/grp.h»	1852
95.7.1	lib/grp/grent.c	1853
95.8	os32: «lib/inttypes.h»	1857

Sorgenti della libreria generale	1791
95.8.1 lib/inttypes/imaxabs.c	1864
95.8.2 lib/inttypes/imaxdiv.c	1865
95.9 os32: «lib/libgen.h»	1865
95.9.1 lib/libgen/basename.c	1866
95.9.2 lib/libgen/dirname.c	1867
95.10 os32: «lib/netinet/icmp.h»	1870
95.11 os32: «lib/netinet/in.h»	1874
95.12 os32: «lib/netinet/ip.h»	1877
95.13 os32: «lib/netinet/tcp.h»	1879
95.14 os32: «lib/netinet/udp.h»	1882
95.15 os32: «lib/pwd.h»	1883
95.15.1 lib/pwd/pwent.c	1884
95.16 os32: «lib/setjmp.h»	1887
95.16.1 lib/setjmp/longjmp.c	1889
95.16.2 lib/setjmp/setjmp.s	1890
95.17 os32: «lib/signal.h»	1891
95.17.1 lib/signal/_sighandler_wrapper.s	1893
95.17.2 lib/signal/kill.c	1895
95.17.3 lib/signal/signal.c	1896
95.18 os32: «lib/stdio.h»	1897
95.18.1 lib/stdio/FILE.c	1902
95.18.2 lib/stdio/clearerr.c	1903

95.18.3	lib/stdio/fclose.c	1903
95.18.4	lib/stdio/feof.c	1903
95.18.5	lib/stdio/ferror.c	1904
95.18.6	lib/stdio/fflush.c	1904
95.18.7	lib/stdio/fgetc.c	1905
95.18.8	lib/stdio/fgetpos.c	1906
95.18.9	lib/stdio/fgets.c	1906
95.18.10	lib/stdio/fileno.c	1908
95.18.11	lib/stdio/fopen.c	1908
95.18.12	lib/stdio/fprintf.c	1910
95.18.13	lib/stdio/fputc.c	1911
95.18.14	lib/stdio/fputs.c	1911
95.18.15	lib/stdio/fread.c	1912
95.18.16	lib/stdio/freopen.c	1913
95.18.17	lib/stdio/fscanf.c	1914
95.18.18	lib/stdio/fseek.c	1915
95.18.19	lib/stdio/fseeko.c	1915
95.18.20	lib/stdio/fsetpos.c	1916
95.18.21	lib/stdio/ftell.c	1917
95.18.22	lib/stdio/ftello.c	1917
95.18.23	lib/stdio/fwrite.c	1917
95.18.24	lib/stdio/getchar.c	1918
95.18.25	lib/stdio/gets.c	1919
95.18.26	lib/stdio/perror.c	1921
95.18.27	lib/stdio/printf.c	1922

95.18.28	lib/stdio/putchar.c	1922
95.18.29	lib/stdio/puts.c	1923
95.18.30	lib/stdio/rewind.c	1923
95.18.31	lib/stdio/scanf.c	1924
95.18.32	lib/stdio/setbuf.c	1924
95.18.33	lib/stdio/setvbuf.c	1924
95.18.34	lib/stdio/snprintf.c	1925
95.18.35	lib/stdio/sprintf.c	1925
95.18.36	lib/stdio/sscanf.c	1926
95.18.37	lib/stdio/vfprintf.c	1926
95.18.38	lib/stdio/vfscanf.c	1927
95.18.39	lib/stdio/vfscanf.c	1928
95.18.40	lib/stdio/vprintf.c	1973
95.18.41	lib/stdio/vscanf.c	1974
95.18.42	lib/stdio/vsnprintf.c	1975
95.18.43	lib/stdio/vsprintf.c	2012
95.18.44	lib/stdio/vsscanf.c	2013
95.19	os32: «lib/stdlib.h»	2013
95.19.1	lib/stdlib/_Exit.c	2017
95.19.2	lib/stdlib/abort.c	2018
95.19.3	lib/stdlib/abs.c	2019
95.19.4	lib/stdlib/atexit.c	2020
95.19.5	lib/stdlib/atoi.c	2021
95.19.6	lib/stdlib/atol.c	2022
95.19.7	lib/stdlib/div.c	2023

95.19.8	lib/stdlib/environment.c	2024
95.19.9	lib/stdlib/exit.c	2026
95.19.10	lib/stdlib/getenv.c	2027
95.19.11	lib/stdlib/labs.c	2029
95.19.12	lib/stdlib/ldiv.c	2030
95.19.13	lib/stdlib/llabs.c	2030
95.19.14	lib/stdlib/lldiv.c	2031
95.19.15	lib/stdlib/putenv.c	2031
95.19.16	lib/stdlib/qsort.c	2034
95.19.17	lib/stdlib/rand.c	2038
95.19.18	lib/stdlib/setenv.c	2039
95.19.19	lib/stdlib/strtol.c	2043
95.19.20	lib/stdlib/strtoul.c	2049
95.19.21	lib/stdlib/unsetenv.c	2049
95.19.22	lib/stdlib_alloc/_alloc_list.c	2052
95.19.23	lib/stdlib_alloc/free.c	2054
95.19.24	lib/stdlib_alloc/malloc.c	2056
95.19.25	lib/stdlib_alloc/realloc.c	2063
95.20	os32: «lib/string.h»	2067
95.20.1	lib/string/memccpy.c	2069
95.20.2	lib/string/memchr.c	2070
95.20.3	lib/string/memcmp.c	2070
95.20.4	lib/string/memcpy.c	2071
95.20.5	lib/string/memmove.c	2071
95.20.6	lib/string/memset.c	2073

95.20.7	lib/string/strcat.c	2073
95.20.8	lib/string/strchr.c	2074
95.20.9	lib/string/stremp.c	2074
95.20.10	lib/string/strcoll.c	2075
95.20.11	lib/string/strcpy.c	2075
95.20.12	lib/string/strcspn.c	2076
95.20.13	lib/string/strdup.c	2077
95.20.14	lib/string/streerror.c	2077
95.20.15	lib/string/strlen.c	2081
95.20.16	lib/string/strncat.c	2082
95.20.17	lib/string/strncmp.c	2082
95.20.18	lib/string/strncpy.c	2083
95.20.19	lib/string/strpbrk.c	2084
95.20.20	lib/string/strrchr.c	2084
95.20.21	lib/string/strspn.c	2085
95.20.22	lib/string/strstr.c	2086
95.20.23	lib/string/strtok.c	2087
95.20.24	lib/string/strxfrm.c	2091
95.21	os32: «lib/sys/os32.h»	2091
95.21.1	lib/sys/os32/input_line.c	2112
95.21.2	lib/sys/os32/ipconfig.c	2116
95.21.3	lib/sys/os32/mount.c	2117
95.21.4	lib/sys/os32/namep.c	2118
95.21.5	lib/sys/os32/routeadd.c	2122
95.21.6	lib/sys/os32/routedel.c	2124

95.21.7	lib/sys/os32/sys.s	2125
95.21.8	lib/sys/os32/umount.c	2125
95.21.9	lib/sys/os32/z_perror.c	2126
95.21.10	lib/sys/os32/z_printf.c	2127
95.21.11	lib/sys/os32/z_vprintf.c	2128
95.22	os32: «lib/sys/sa_family_t.h»	2128
95.23	os32: «lib/sys/socket.h»	2129
95.23.1	lib/sys/socket/accept.c	2131
95.23.2	lib/sys/socket/bind.c	2133
95.23.3	lib/sys/socket/connect.c	2134
95.23.4	lib/sys/socket/listen.c	2136
95.23.5	lib/sys/socket/recvfrom.c	2137
95.23.6	lib/sys/socket/send.c	2140
95.23.7	lib/sys/socket/socket.c	2142
95.24	os32: «lib/sys/socklen_t.h»	2143
95.25	os32: «lib/sys/stat.h»	2144
95.25.1	lib/sys/stat/chmod.c	2148
95.25.2	lib/sys/stat/fchmod.c	2149
95.25.3	lib/sys/stat/fstat.c	2150
95.25.4	lib/sys/stat/mkdir.c	2151
95.25.5	lib/sys/stat/mknod.c	2152
95.25.6	lib/sys/stat/stat.c	2152
95.25.7	lib/sys/stat/umask.c	2154
95.26	os32: «lib/sys/types.h»	2154

95.26.1	lib/sys/types/major.c	2155
95.26.2	lib/sys/types/makedev.c	2156
95.26.3	lib/sys/types/minor.c	2156
95.27	os32: «lib/sys/wait.h»	2156
95.27.1	lib/sys/wait/wait.c	2157
95.28	os32: «lib/termios.h»	2158
95.28.1	lib/termios/tcgetattr.c	2161
95.28.2	lib/termios/tcsetattr.c	2161
95.29	os32: «lib/time.h»	2162
95.29.1	lib/time/asctime.c	2164
95.29.2	lib/time/clock.c	2166
95.29.3	lib/time/gmtime.c	2167
95.29.4	lib/time/mktime.c	2172
95.29.5	lib/time/stime.c	2176
95.29.6	lib/time/time.c	2177
95.30	os32: «lib/unistd.h»	2177
95.30.1	lib/unistd/_exit.c	2182
95.30.2	lib/unistd/access.c	2183
95.30.3	lib/unistd/brk.c	2184
95.30.4	lib/unistd/chdir.c	2185
95.30.5	lib/unistd/chown.c	2186
95.30.6	lib/unistd/close.c	2187
95.30.7	lib/unistd/dup.c	2187

95.30.8	lib/unistd/dup2.c	2188
95.30.9	lib/unistd/environ.c	2189
95.30.10	lib/unistd/execl.c	2189
95.30.11	lib/unistd/execl.e.c	2190
95.30.12	lib/unistd/execlp.c	2191
95.30.13	lib/unistd/execv.c	2193
95.30.14	lib/unistd/execve.c	2193
95.30.15	lib/unistd/execvp.c	2196
95.30.16	lib/unistd/fchdir.c	2197
95.30.17	lib/unistd/fchown.c	2197
95.30.18	lib/unistd/fork.c	2198
95.30.19	lib/unistd/getcwd.c	2199
95.30.20	lib/unistd/getegid.c	2201
95.30.21	lib/unistd/geteuid.c	2201
95.30.22	lib/unistd/getgid.c	2202
95.30.23	lib/unistd/getopt.c	2202
95.30.24	lib/unistd/getpgrp.c	2209
95.30.25	lib/unistd/getpid.c	2210
95.30.26	lib/unistd/getppid.c	2210
95.30.27	lib/unistd/getuid.c	2211
95.30.28	lib/unistd/isatty.c	2211
95.30.29	lib/unistd/link.c	2213
95.30.30	lib/unistd/lseek.c	2213
95.30.31	lib/unistd/pipe.c	2214
95.30.32	lib/unistd/read.c	2215

95.30.33	lib/unistd/rmdir.c	2218
95.30.34	lib/unistd/sbrk.c	2219
95.30.35	lib/unistd/setegid.c	2220
95.30.36	lib/unistd/seteuid.c	2220
95.30.37	lib/unistd/setgid.c	2221
95.30.38	lib/unistd/setpgrp.c	2222
95.30.39	lib/unistd/setuid.c	2222
95.30.40	lib/unistd/sleep.c	2223
95.30.41	lib/unistd/ttyname.c	2224
95.30.42	lib/unistd/unlink.c	2226
95.30.43	lib/unistd/write.c	2226
95.31	os32: «lib/utime.h»	2228
95.31.1	lib/utime/utime.c	2229

95.1 os32: file isolati della directory «lib/»

95.1.1 lib/NULL.h

Si veda la sezione [91.3](#).

```
3150001 #ifndef _NULL_H
3150002 #define _NULL_H      1
3150003 //-----
3150004 #define NULL ((void *) 0)
3150005 //-----
3150006 #endif
```

95.1.2 lib/SEEK.h



Si veda la sezione [91.3](#).

```

3160001 #ifndef _SEEK_H
3160002 #define _SEEK_H      1
3160003 //-----
3160004 // These values are used inside 'stdio.h' and
3160005 // 'unistd.h'.
3160006 //-----
3160007 #define SEEK_SET      0      // From the start.
3160008 #define SEEK_CUR      1      // From current
3160009                          // position.
3160010 #define SEEK_END      2      // From the end.
3160011 //-----
3160012 #endif

```

95.1.3 lib/assert.h



Si veda la sezione [88.6](#).

```

3170001 #ifndef _ASSERT_H
3170002 #define _ASSERT_H      1
3170003 //-----
3170004 #include <stdio.h>
3170005 //-----
3170006 #ifdef NDEBUG
3170007 #define assert(ignore) ((void)0)
3170008 #else
3170009 #define assert(ASSERTION) \
3170010     ({if ((ASSERTION)==0) \
3170011         fprintf (stderr, \
3170012             "Assertion failed: " # ASSERTION \
3170013             ", function %s, file %s, line %u.\n", \
3170014             __func__, __FILE__, __LINE__);})
3170015 #endif
3170016 //-----
3170017 #endif

```


95.1.4 lib/clock_t.h



Si veda la sezione [91.3](#).

```
3180001 #ifndef _CLOCK_T_H
3180002 #define _CLOCK_T_H      1
3180003 //-----
3180004 #include <stdint.h>
3180005 //-----
3180006 typedef uint64_t clock_t;
3180007 //-----
3180008 #endif
```

95.1.5 lib/ctype.h



Si veda la sezione [91.3](#).

```
3190001 #ifndef _CTYPE_H
3190002 #define _CTYPE_H      1
3190003 //-----
3190004 #include <NULL.h>
3190005 //-----
3190006 #define isblank(C)  ((int) (C == ' ' || C == '\t'))
3190007 #define isspace(C)  ((int) (C == ' ' \
3190008                       || C == '\f' \
3190009                       || C == '\n' \
3190010                       || C == '\r' \
3190011                       || C == '\t' \
3190012                       || C == '\v'))
3190013 #define isdigit(C)  ((int) (C >= '0' && C <= '9'))
3190014 #define isxdigit(C) \
3190015     ((int) ((C >= '0' && C <= '9') \
3190016           || (C >= 'A' && C <= 'F') \
3190017           || (C >= 'a' && C <= 'f')))
3190018 #define isupper(C)  ((int) (C >= 'A' && C <= 'Z'))
3190019 #define islower(C)  ((int) (C >= 'a' && C <= 'z'))
3190020 #define iscntrl(C)  ((int) ((C >= 0x00 && C <= 0x1F) \
```

```
3190022         || C == 0x7F))
3190023 #define isgraph(C) ((int) (C >= 0x21 && C <= 0x7E))
3190024 #define isprint(C) ((int) (C >= 0x20 && C <= 0x7E))
3190025 #define isalpha(C) (isupper (C) || islower (C))
3190026 #define isalnum(C) (isalpha (C) || isdigit (C))
3190027 #define ispunct(C) (isgraph (C) && (!isspace (C)) \
3190028         && (!isalnum (C)))
3190029 #define tolower(C) (isupper (C) ? ((C) + 0x20) : (C))
3190030 #define toupper(C) (islower (C) ? ((C) - 0x20) : (C))
3190031 #define toascii(C) (C & 0x7F)
3190032 #define _tolower(C) (isupper (C) ? ((C) + 0x20) : (C))
3190033 #define _toupper(C) (islower (C) ? ((C) - 0x20) : (C))
3190034 //-----
3190035 #endif
```

95.1.6 lib/limits.h

«

Si veda la sezione [91.3](#).

```
3200001 #ifndef _LIMITS_H
3200002 #define _LIMITS_H      1
3200003 //-----
3200004 #define CHAR_UNSIGNED  0
3200005 //-----
3200006 #define CHAR_BIT      (8)
3200007 //
3200008 #define SCHAR_MIN     (-0x80)
3200009 #define SCHAR_MAX     (0x7F)
3200010 #define UCHAR_MAX     (0xFF)
3200011 //
3200012 #ifdef CHAR_UNSIGNED
3200013 #define CHAR_MIN      (0)
3200014 #define CHAR_MAX      UCHAR_MAX
3200015 #else
3200016 #define CHAR_MIN      SCHAR_MIN
3200017 #define CHAR_MAX      SCHAR_MAX
3200018 #endif
```



```

3200056 #define MAX_INPUT      1          // Max bytes in tty
3200057                                     // input queue.
3200058 //-----
3200059 #define CHLD_MAX      INT_MAX    // Not used.
3200060 #define HOST_NAME_MAX INT_MAX    // Not used.
3200061 #define LOGIN_NAME_MAX INT_MAX   // Not used.
3200062 #define PAGE_SIZE     INT_MAX    // Not used.
3200063 #define RE_DUP_MAX    INT_MAX    // Not used.
3200064 #define STREAM_MAX    INT_MAX    // Not used.
3200065 #define SYMLOOP_MAX   INT_MAX    // Not used.
3200066 #define TTY_NAME_MAX  INT_MAX    // Not used.
3200067 #define TZNAME_MAX    INT_MAX    // Not used.
3200068 #define PIPE_MAX      INT_MAX    // Not used.
3200069 #define SYMLINK_MAX   INT_MAX    // Not used.
3200070 //-----
3200071 #endif

```

95.1.7 lib/ptrdiff_t.h



Si veda la sezione [91.3](#).

```

3210001 #ifndef _PTRDIFF_T_H
3210002 #define _PTRDIFF_T_H      1
3210003 //-----
3210004 typedef int ptrdiff_t;
3210005 //-----
3210006 #endif

```

95.1.8 lib/restrict.h



Si veda la sezione [91.3](#).

```

3220001 #ifndef _RESTRICT_H
3220002 #define _RESTRICT_H      1
3220003 //-----
3220004 // At the moment, the GCC compiler does not support
3220005 // the 'restrict' keyword.

```

```

3220006 //-----
3220007 #define restrict /**/
3220008 //-----
3220009 #endif

```

95.1.9 lib/size_t.h

Si veda la sezione [91.3](#).

```

3230001 #ifndef _SIZE_T_H
3230002 #define _SIZE_T_H          1
3230003 //-----
3230004 // The type 'size_t' *must* be equal to an 'int'.
3230005 //-----
3230006 typedef unsigned int size_t;
3230007 //-----
3230008 #endif

```

95.1.10 lib/stdarg.h

Si veda la sezione [91.3](#).

```

3240001 #ifndef _STDARG_H
3240002 #define _STDARG_H          1
3240003 //-----
3240004 typedef unsigned char *va_list;
3240005 //-----
3240006 #define va_start(ap, last) \
3240007     ((void) ((ap) = \
3240008         ((va_list) &(last)) + (sizeof (last))))
3240009 #define va_end(ap) ((void) ((ap) = 0))
3240010 #define va_copy(dest, src) \
3240011     ((void) ((dest) = (va_list) (src)))
3240012 #define va_arg(ap, type) \
3240013     (((ap) = (ap) + (sizeof (type))), \
3240014     *((type *) ((ap) - (sizeof (type)))))
3240015 //-----

```

3240016	#endif
---------	--------

95.1.11 lib/stdbool.h



Si veda la sezione [91.3](#).

3250001	#ifndef _STDBOOL_H
3250002	#define _STDBOOL_H 1
3250003	//-----
3250004	#define bool _Bool
3250005	#define true 1
3250006	#define false 0
3250007	#define __bool_true_false_are_defined 1
3250008	//-----
3250009	#endif

95.1.12 lib/stddef.h



Si veda la sezione [91.3](#).

3260001	#ifndef _STDDEF_H
3260002	#define _STDDEF_H 1
3260003	//-----
3260004	#include <ptrdiff_t.h>
3260005	#include <size_t.h>
3260006	#include <wchar_t.h>
3260007	#include <NULL.h>
3260008	//-----
3260009	#define offsetof(type, member) \
3260010	((size_t) &((type *)0)->member)
3260011	//-----
3260012	#endif

95.1.13 lib/stdint.h



Si veda la sezione [91.3](#).

```
3270001 #ifndef _STDINT_H
3270002 #define _STDINT_H          1
3270003 //-----
3270004 typedef signed char int8_t;
3270005 typedef short int int16_t;
3270006 typedef int int32_t;
3270007 typedef long long int int64_t;
3270008 //
3270009 typedef unsigned char uint8_t;
3270010 typedef unsigned short int uint16_t;
3270011 typedef unsigned int uint32_t;
3270012 typedef unsigned long long int uint64_t;
3270013 //
3270014 #define INT8_MIN            (-0x80)
3270015 #define INT16_MIN           (-0x8000)
3270016 #define INT32_MIN           (-0x80000000)
3270017 #define INT64_MIN           (-0x8000000000000000LL)
3270018 //
3270019 #define INT8_MAX            0x7F
3270020 #define INT16_MAX           0x7FFF
3270021 #define INT32_MAX           0x7FFFFFFF
3270022 #define INT64_MAX           0x7FFFFFFFFFFFFFFFLL
3270023 //
3270024 #define UINT8_MAX           0xFF
3270025 #define UINT16_MAX          0xFFFF
3270026 #define UINT32_MAX          0xFFFFFFFFU
3270027 #define UINT64_MAX          0xFFFFFFFFFFFFFFFFULL
3270028 //-----
3270029 typedef signed char int_least8_t;
3270030 typedef short int int_least16_t;
3270031 typedef int int_least32_t;
3270032 typedef long long int int_least64_t;
3270033 //
3270034 typedef unsigned char uint_least8_t;
```

```

3270035 typedef unsigned short int uint_least16_t;
3270036 typedef unsigned int uint_least32_t;
3270037 typedef unsigned long long int uint_least64_t;
3270038 //
3270039 #define INT_LEAST8_MIN (-0x80)
3270040 #define INT_LEAST16_MIN (-0x8000)
3270041 #define INT_LEAST32_MIN (-0x80000000)
3270042 #define INT_LEAST64_MIN (-0x8000000000000000LL)
3270043 //
3270044 #define INT_LEAST8_MAX 0x7F
3270045 #define INT_LEAST16_MAX 0x7FFF
3270046 #define INT_LEAST32_MAX 0x7FFFFFFF
3270047 #define INT_LEAST64_MAX 0x7FFFFFFFFFFFFFFFFFLL
3270048 //
3270049 #define UINT_LEAST8_MAX 0xFF
3270050 #define UINT_LEAST16_MAX 0xFFFF
3270051 #define UINT_LEAST32_MAX 0xFFFFFFFFU
3270052 #define UINT_LEAST64_MAX 0xFFFFFFFFFFFFFFFFULL
3270053 //-----
3270054 #define INT8_C (VAL) VAL
3270055 #define INT16_C (VAL) VAL
3270056 #define INT32_C (VAL) VAL
3270057 #define INT64_C (VAL) VAL ## LL
3270058 //
3270059 #define UINT8_C (VAL) VAL
3270060 #define UINT16_C (VAL) VAL
3270061 #define UINT32_C (VAL) VAL ## U
3270062 #define UINT64_C (VAL) VAL ## ULL
3270063 //-----
3270064 typedef signed char int_fast8_t;
3270065 typedef int int_fast16_t;
3270066 typedef int int_fast32_t;
3270067 typedef long long int int_fast64_t;
3270068 //
3270069 typedef unsigned char uint_fast8_t;
3270070 typedef unsigned int uint_fast16_t;
3270071 typedef unsigned int uint_fast32_t;

```



```
3270072 typedef unsigned long long int uint_fast64_t;
3270073 //
3270074 #define INT_FAST8_MIN (-0x80)
3270075 #define INT_FAST16_MIN (-0x80000000)
3270076 #define INT_FAST32_MIN (-0x80000000)
3270077 #define INT_FAST64_MIN (-0x8000000000000000LL)
3270078 //
3270079 #define INT_FAST8_MAX 0x7F
3270080 #define INT_FAST16_MAX 0x7FFFFFFF
3270081 #define INT_FAST32_MAX 0x7FFFFFFF
3270082 #define INT_FAST64_MAX 0x7FFFFFFFFFFFFFFFLL
3270083 //
3270084 #define UINT_FAST8_MAX 0xFF
3270085 #define UINT_FAST16_MAX 0xFFFFFFFFU
3270086 #define UINT_FAST32_MAX 0xFFFFFFFFU
3270087 #define UINT_FAST64_MAX 0xFFFFFFFFFFFFFFFFULL
3270088 //-----
3270089 typedef int intptr_t;
3270090 typedef unsigned int uintptr_t;
3270091 //
3270092 #define INTPTR_MIN (-0x80000000)
3270093 #define INTPTR_MAX 0x7FFFFFFF
3270094 #define UINTPTR_MAX 0xFFFFFFFFU
3270095 //
3270096 typedef long long int intmax_t;
3270097 typedef unsigned long long int uintmax_t;
3270098 //
3270099 #define INTMAX_C(VAL) VAL ## LL
3270100 #define UINTMAX_C(VAL) VAL ## ULL
3270101 #define INTMAX_MIN (-INTMAX_C(0x8000000000000000))
3270102 #define INTMAX_MAX (INTMAX_C(0x7FFFFFFFFFFFFFFF))
3270103 #define UINTMAX_MAX (UINTMAX_C(0xFFFFFFFFFFFFFFFF))
3270104 //-----
3270105 #define PTRDIFF_MIN (-0x80000000)
3270106 #define PTRDIFF_MAX 0x7FFFFFFF
3270107 //
3270108 #define SIG_ATOMIC_MIN (-0x80000000)
```

```

3270109 #define SIG_ATOMIC_MAX          0x7FFFFFFF
3270110 //
3270111 #define SIZE_MAX                0xFFFFFFFFU
3270112 //
3270113 #define WCHAR_MIN              0x00000000
3270114 #define WCHAR_MAX              0xFFFFFFFFU
3270115 //
3270116 #define WINT_MIN                (-0x8000000000000000LL)
3270117 #define WINT_MAX                0x7FFFFFFFFFFFFFFFLL
3270118 //-----
3270119 #endif

```

95.1.14 lib/time_t.h

«

Si veda la sezione [91.3](#).

```

3280001 #ifndef _TIME_T_H
3280002 #define _TIME_T_H          1
3280003 //-----
3280004 typedef long long int time_t;
3280005 //-----
3280006 #endif

```

95.1.15 lib/wchar_t.h

«

Si veda la sezione [91.3](#).

```

3290001 #ifndef _WCHAR_T_H
3290002 #define _WCHAR_T_H        1
3290003 //-----
3290004 typedef unsigned int wchar_t;
3290005 //-----
3290006 #endif

```

95.2 os32: «lib/_gcc.h»



Si veda la sezione [88.1](#).

```

3300001 #ifndef __GCC_H
3300002 #define __GCC_H          1
3300003 //-----
3300004 #include <stdlib.h>
3300005 //-----
3300006 typedef struct
3300007 {
3300008     unsigned long long int quot;
3300009     unsigned long long int rem;
3300010 } ulldiv_t;
3300011 //-----
3300012 lldiv_t _lldiv (long long int dividend,
3300013                long long int divisor);
3300014 ulldiv_t _ulldiv (unsigned long long int dividend,
3300015                  unsigned long long int divisor);
3300016 //-----
3300017 unsigned long long int __udivdi3 (unsigned long long
3300018                                  int dividend,
3300019                                  unsigned long long
3300020                                  int divisor);
3300021 unsigned long long int __umoddi3 (unsigned long long
3300022                                  int dividend,
3300023                                  unsigned long long
3300024                                  int divisor);
3300025 long long int __divdi3 (long long int dividend,
3300026                        long long int divisor);
3300027 long long int __moddi3 (long long int dividend,
3300028                        long long int divisor);
3300029 //-----
3300030 #endif

```

[95.2.1](#) lib/_gcc/__divdi3.c 1812

[95.2.2](#) lib/_gcc/__moddi3.c 1812

95.2.3	lib/_gcc/___udivdi3.c	1813
95.2.4	lib/_gcc/___umoddi3.c	1813
95.2.5	lib/_gcc/___lldiv.c	1813
95.2.6	lib/_gcc/___ulldiv.c	1815

95.2.1 lib/_gcc/___divdi3.c

«

Si veda la sezione [88.1](#).

```

3310001 #include <_gcc.h>
3310002 //-----
3310003 long long int
3310004 ___divdi3 (long long int dividend, long long int divisor)
3310005 {
3310006     lldiv_t result;
3310007     result = _lldiv (dividend, divisor);
3310008     return result.quot;
3310009 }
```

95.2.2 lib/_gcc/___moddi3.c

«

Si veda la sezione [88.1](#).

```

3320001 #include <_gcc.h>
3320002 //-----
3320003 long long int
3320004 ___moddi3 (long long int dividend, long long int divisor)
3320005 {
3320006     lldiv_t result;
3320007     result = _lldiv (dividend, divisor);
3320008     return result.rem;
3320009 }
```

95.2.3 lib/_gcc/__udivdi3.c



Si veda la sezione [88.1](#).

```
3330001 #include <_gcc.h>
3330002 //-----
3330003 unsigned long long int
3330004 __udivdi3 (unsigned long long int dividend,
3330005           unsigned long long int divisor)
3330006 {
3330007     ulldiv_t result;
3330008     result = _udiv (dividend, divisor);
3330009     return result.quot;
3330010 }
```

95.2.4 lib/_gcc/__umoddi3.c



Si veda la sezione [88.1](#).

```
3340001 #include <_gcc.h>
3340002 //-----
3340003 unsigned long long int
3340004 __umoddi3 (unsigned long long int dividend,
3340005           unsigned long long int divisor)
3340006 {
3340007     ulldiv_t result;
3340008     result = _udiv (dividend, divisor);
3340009     return result.rem;
3340010 }
```

95.2.5 lib/_gcc/_lldiv.c



Si veda la sezione [88.1](#).

```
3350001 #include <_gcc.h>
3350002 //-----
3350003 // If DIVIDEND and DIVISOR have different sign,
3350004 // the QUOTIENT is negative.
```

```
3350005 //
3350006 // The REMINDER has the same sign as the DIVISOR.
3350007 //-----
3350008 lldiv_t
3350009 _lldiv (long long int dividend, long long int divisor)
3350010 {
3350011     ulldiv_t uresult;
3350012     lldiv_t result;
3350013     //
3350014     // Check for sign.
3350015     //
3350016     if (dividend >= 0 && divisor >= 0)
3350017     {
3350018         uresult = _ulldiv ((unsigned long long) dividend,
3350019                         (unsigned long long) divisor);
3350020         result.quot = uresult.quot;
3350021         result.rem = uresult.rem;
3350022     }
3350023     else if (dividend < 0 && divisor < 0)
3350024     {
3350025         uresult =
3350026             _ulldiv ((unsigned long long) -dividend,
3350027                     (unsigned long long) -divisor);
3350028         result.quot = uresult.quot;
3350029         result.rem = -uresult.rem;
3350030     }
3350031     else if (dividend < 0 && divisor >= 0)
3350032     {
3350033         uresult =
3350034             _ulldiv ((unsigned long long) -dividend,
3350035                     (unsigned long long) divisor);
3350036         result.quot = -uresult.quot;
3350037         result.rem = uresult.rem;
3350038     }
3350039     else if (dividend >= 0 && divisor < 0)
3350040     {
3350041         uresult = _ulldiv ((unsigned long long) dividend,
```

```
3350042         (unsigned long long) -divisor);
3350043     result.quot = uresult.quot;
3350044     result.rem = -uresult.rem;
3350045 }
3350046 //
3350047 return (result);
3350048 }
```

95.2.6 lib/_gcc/_ulldiv.c

Si veda la sezione [88.1](#).

```
3360001 #include <_gcc.h>
3360002 //-----
3360003 // DIVIDEND = DIVISOR * QUOTIENT + REMINDER
3360004 //
3360005 // If DIVISOR == 0,
3360006 // then QUOTIENT == 0 and REMINDER == DIVIDEND
3360007 //-----
3360008 ulldiv_t
3360009 _ulldiv (unsigned long long int dividend,
3360010         unsigned long long int divisor)
3360011 {
3360012     unsigned long long int sign;
3360013     unsigned long long int mask;
3360014     ulldiv_t result;
3360015     int scroll;
3360016     unsigned int size;    // Bits of a long long.
3360017     //
3360018     // Division of zero will return zero.
3360019     //
3360020     if (dividend == 0)
3360021     {
3360022         result.quot = 0;
3360023         result.rem = 0;
3360024         return (result);
3360025     }
```

```
3360026 //
3360027 // Division by zero will return zero and all
3360028 // remainder.
3360029 //
3360030 if (divisor == 0)
3360031 {
3360032     result.quot = 0;
3360033     result.rem = dividend;
3360034     return (result);
3360035 }
3360036 //
3360037 // Calculate how much bits does have the type 'long
3360038 // long'.
3360039 //
3360040 size = 0;
3360041 mask = ~0LL;
3360042 //
3360043 while (mask > 0)
3360044 {
3360045     size += 8;
3360046     mask >>= 8;
3360047 }
3360048 //
3360049 // Calculate the value for 'sign' that needs to have
3360050 // the most
3360051 // significant bit to one.
3360052 //
3360053 mask = ~0LL;
3360054 mask >>= 1;
3360055 sign = ~mask;
3360056 //
3360057 // Scroll divisor to the left, as long as the first
3360058 // bit is zero.
3360059 //
3360060 for (scroll = 0; scroll < size; scroll++)
3360061 {
3360062     if (divisor & sign)
```



```
3360063     {
3360064         //
3360065         // The most significant bit is one.
3360066         //
3360067         break;
3360068     }
3360069     //
3360070     // The most significant bit is zero: scroll
3360071     // left.
3360072     //
3360073     divisor <<= 1;
3360074 }
3360075 //
3360076 //
3360077 //
3360078 result.quot = 0;
3360079 result.rem = 0;
3360080 //
3360081 for (; scroll >= 0 && divisor > 0; scroll--)
3360082     {
3360083         result.quot <<= 1;
3360084         if (dividend >= divisor)
3360085             {
3360086                 result.quot |= 1LL;
3360087                 dividend -= divisor;
3360088             }
3360089         divisor >>= 1;
3360090     }
3360091 //
3360092 result.rem = dividend;
3360093 //
3360094 return (result);
3360095 }
```

95.3 os32: «lib/arpa/inet.h»

«

Si veda la sezione [91.3](#).

```

3370001 #ifndef _ARPA_INET_H
3370002 #define _ARPA_INET_H      1
3370003 //-----
3370004 #include <stdint.h>
3370005 #include <sys/socklen_t.h>
3370006 //-----
3370007 uint32_t htonl (uint32_t host32);
3370008 uint16_t htons (uint16_t host16);
3370009 uint32_t ntohl (uint32_t net32);
3370010 uint16_t ntohs (uint16_t net16);
3370011 //-----
3370012 const char *inet_ntop (int family, const void *src,
3370013                       char *dst, socklen_t size);
3370014 int inet_pton (int family, const char *src, void *dst);
3370015 //-----
3370016 #endif

```

95.3.1	lib/arpa/inet/htonl.c	1819
95.3.2	lib/arpa/inet/htons.c	1819
95.3.3	lib/arpa/inet/inet_ntop.c	1820
95.3.4	lib/arpa/inet/inet_pton.c	1821
95.3.5	lib/arpa/inet/ntohl.c	1824
95.3.6	lib/arpa/inet/ntohs.c	1825

95.3.1 lib/arpa/inet/htonl.c



Si veda la sezione [88.11](#).

```
3380001 #include <arpa/inet.h>
3380002 //-----
3380003 uint32_t
3380004 htonl (uint32_t host32)
3380005 {
3380006     uint8_t *orig = (void *) &host32;
3380007     union
3380008     {
3380009         uint32_t value;
3380010         uint8_t b[4];
3380011     } dest;
3380012     //
3380013     // Convert: must revert byte order.
3380014     //
3380015     dest.b[0] = orig[3];
3380016     dest.b[1] = orig[2];
3380017     dest.b[2] = orig[1];
3380018     dest.b[3] = orig[0];
3380019     //
3380020     return (dest.value);
3380021 }
```

95.3.2 lib/arpa/inet/htons.c



Si veda la sezione [88.11](#).

```
3390001 #include <arpa/inet.h>
3390002 //-----
3390003 uint16_t
3390004 htons (uint16_t host16)
3390005 {
3390006     uint8_t *orig = (void *) &host16;
3390007     union
3390008     {
```

```
3390009     uint16_t value;
3390010     uint8_t b[2];
3390011 } dest;
3390012 //
3390013 // Convert: must revert byte order.
3390014 //
3390015 dest.b[0] = orig[1];
3390016 dest.b[1] = orig[0];
3390017 //
3390018 return (dest.value);
3390019 }
```

95.3.3 lib/arpa/inet/inet_ntop.c

«

Si veda la sezione [88.66](#).

```
3400001 #include <arpa/inet.h>
3400002 #include <stdint.h>
3400003 #include <errno.h>
3400004 #include <string.h>
3400005 #include <stdlib.h>
3400006 //-----
3400007 const char *
3400008 inet_ntop (int family, const void *src, char *dst,
3400009           socklen_t size)
3400010 {
3400011 //
3400012 // Check family type: only IPv4 is available here.
3400013 //
3400014 if (family != AF_INET)
3400015     {
3400016         errset (EAFNOSUPPORT);
3400017         return (NULL);
3400018     }
3400019 //
3400020 // Check for NULL pointers.
3400021 //
```

```
3400022     if (src == NULL || dst == NULL)
3400023     {
3400024         errset (EINVAL);
3400025         return (NULL);
3400026     }
3400027     //
3400028     snprintf (dst, (size_t) size, "%i.%i.%i.%i",
3400029              *((in_addr_t *) src) >> 0 & 0x000000FF,
3400030              *((in_addr_t *) src) >> 8 & 0x000000FF,
3400031              *((in_addr_t *) src) >> 16 & 0x000000FF,
3400032              *((in_addr_t *) src) >> 24 & 0x000000FF);
3400033     //
3400034     // Return ok.
3400035     //
3400036     return (dst);
3400037 }
```

95.3.4 lib/arpa/inet/inet_pton.c

Si veda la sezione [88.67](#).

```
3410001 #include <arpa/inet.h>
3410002 #include <stdint.h>
3410003 #include <errno.h>
3410004 #include <string.h>
3410005 #include <stdlib.h>
3410006 //-----
3410007 #define INET_PTON_MAX_STRING_SIZE 31
3410008 //-----
3410009 int
3410010 inet_pton (int family, const char *src, void *dst)
3410011 {
3410012     char *t;
3410013     int ipv4[4];
3410014     int i;
3410015     in_addr_t result;
3410016     char source[INET_PTON_MAX_STRING_SIZE + 1];
```

```
3410017 //
3410018 // Check family type: only IPv4 is available here.
3410019 //
3410020 if (family != AF_INET)
3410021 {
3410022     errset (EAFNOSUPPORT);
3410023     return (-1);
3410024 }
3410025 //
3410026 // Check for NULL pointers.
3410027 //
3410028 if (src == NULL || dst == NULL)
3410029 {
3410030     errset (EINVAL);
3410031     return (-1);
3410032 }
3410033 //
3410034 // Check the source string size.
3410035 //
3410036 if (strlen (src) > INET_PTON_MAX_STRING_SIZE)
3410037 {
3410038     //
3410039     // The IPv4 address scan is finished
3410040     // prematurely:
3410041     // return zero to tell that the address string
3410042     // is
3410043     // not correct.
3410044     //
3410045     return (0);
3410046 }
3410047 //
3410048 // Copy the source address, to be able to modify
3410049 // the string.
3410050 //
3410051 strcpy (source, src);
3410052 //
3410053 // Start ``tokenize`` the string: it is here
```

```
3410054 // accepted also
3410055 // the space as a delimiter.
3410056 //
3410057 t = strtok (source, ". ");
3410058 //
3410059 for (i = 0; i < 4 && t != NULL; i++)
3410060 {
3410061     ipv4[i] = atoi (t);
3410062     if (ipv4[i] > 255 || ipv4[i] < 0)
3410063     {
3410064         //
3410065         // An octet cannot have a value greater than
3410066         // 255,
3410067         // and cannot be negative.
3410068         //
3410069         break;
3410070     }
3410071     t = strtok (NULL, ". ");
3410072 }
3410073 //
3410074 if (i < 4)
3410075 {
3410076     //
3410077     // The IPv4 address scan is finished
3410078     // prematurely:
3410079     // return zero to tell that the address string
3410080     // is
3410081     // not correct.
3410082     //
3410083     return (0);
3410084 }
3410085 //
3410086 // Translate into a network byte order IPv4 address:
3410087 // the architecture is little-endian.
3410088 //
3410089 result = 0;
3410090 result += (ipv4[0] << 0) & 0x000000FF;
```

```
3410091     result += (ipv4[1] << 8) & 0x0000FF00;
3410092     result += (ipv4[2] << 16) & 0x00FF0000;
3410093     result += (ipv4[3] << 24) & 0xFF000000;
3410094     //
3410095     // Update the destination.
3410096     //
3410097     *((in_addr_t *) dst) = result;
3410098     //
3410099     // Return ok.
3410100     //
3410101     return (1);
3410102 }
```

95.3.5 lib/arpa/inet/ntohl.c



Si veda la sezione [88.11](#).

```
3420001 #include <arpa/inet.h>
3420002 //-----
3420003 uint32_t
3420004 ntohl (uint32_t net32)
3420005 {
3420006     uint8_t *orig = (void *) &net32;
3420007     union
3420008     {
3420009         uint32_t value;
3420010         uint8_t b[4];
3420011     } dest;
3420012     //
3420013     // Convert: must revert byte order.
3420014     //
3420015     dest.b[0] = orig[3];
3420016     dest.b[1] = orig[2];
3420017     dest.b[2] = orig[1];
3420018     dest.b[3] = orig[0];
3420019     //
3420020     return (dest.value);
```



```
3420021 }  
}
```

95.3.6 lib/arpa/inet/ntohs.c

Si veda la sezione [88.11](#).

```
3430001 #include <arpa/inet.h>  
3430002 //-----  
3430003 uint16_t  
3430004 ntohs (uint16_t net16)  
3430005 {  
3430006     uint8_t *orig = (void *) &net16;  
3430007     union  
3430008     {  
3430009         uint16_t value;  
3430010         uint8_t b[2];  
3430011     } dest;  
3430012     //  
3430013     // Convert: must revert byte order.  
3430014     //  
3430015     dest.b[0] = orig[1];  
3430016     dest.b[1] = orig[0];  
3430017     //  
3430018     return (dest.value);  
3430019 }
```

95.4 os32: «lib/dirent.h»

Si veda la sezione [91.3](#).

```
3440001 #ifndef _DIRENT_H  
3440002 #define _DIRENT_H        1  
3440003  
3440004 #include <sys/types.h>  // ino_t  
3440005 #include <limits.h>    // NAME_MAX  
3440006
```

```
3440007 //-----
3440008 struct dirent
3440009 {
3440010     ino_t d_ino; // I-node number [1]
3440011     char d_name[NAME_MAX + 1]; // NAME_MAX + Null
3440012     // termination
3440013 } __attribute__ ((packed));
3440014 //
3440015 // [1] The type 'ino_t' must be equal to 'uint16_t',
3440016 //     because the directory inside the Minix 1 file
3440017 //     system has exactly such size.
3440018 //
3440019 //-----
3440020 #define DOPEN_MAX OPEN_MAX/2 // <limits.h> [1]
3440021 //
3440022 // [1] DOPEN_MAX is not standard, but it is used to
3440023 //     define how many directory slot to keep for open
3440024 //     directories. As directory streams are opened as
3440025 //     file descriptors, the sum of all kind of file
3440026 //     open cannot be more than OPEM_MAX.
3440027 //-----
3440028 typedef struct
3440029 {
3440030     int fdn; // File descriptor number.
3440031     struct dirent dir; // Last directory item read.
3440032 } DIR;
3440033
3440034 extern DIR _directory_stream[]; // Defined inside
3440035 //                               // 'lib/dirent/DIR.c'.
3440036 //-----
3440037 // Function prototypes.
3440038 //-----
3440039 int closedir (DIR * dp);
3440040 DIR *opendir (const char *name);
3440041 struct dirent *readdir (DIR * dp);
3440042 void rewinddir (DIR * dp);
3440043 //-----
```

3440044	
3440045	#endif

95.4.1	lib/dirent/DIR.c	1827
95.4.2	lib/dirent/closedir.c	1828
95.4.3	lib/dirent/opendir.c	1829
95.4.4	lib/dirent/readdir.c	1832
95.4.5	lib/dirent/rewinddir.c	1834

95.4.1 lib/dirent/DIR.c

Si veda la sezione [91.3](#).



```
3450001 #include <dirent.h>
3450002 //
3450003 // There must be room for at least 'DOPEN_MAX'
3450004 // elements.
3450005 //
3450006 DIR _directory_stream[DOPEN_MAX];
3450007
3450008 void
3450009 _dirent_directory_stream_setup (void)
3450010 {
3450011     int d;
3450012     //
3450013     for (d = 0; d < DOPEN_MAX; d++)
3450014     {
3450015         _directory_stream[d].fdn = -1;
3450016     }
3450017 }
```

95.4.2 lib/dirent/closedir.c



Si veda la sezione [88.13](#).

```
3460001 #include <dirent.h>
3460002 #include <fcntl.h>
3460003 #include <sys/types.h>
3460004 #include <sys/stat.h>
3460005 #include <unistd.h>
3460006 #include <errno.h>
3460007 #include <stddef.h>
3460008 //-----
3460009 int
3460010 closedir (DIR * dp)
3460011 {
3460012     //
3460013     // Check for a valid argument
3460014     //
3460015     if (dp == NULL)
3460016     {
3460017         //
3460018         // Not a valid pointer.
3460019         //
3460020         errset (EBADF); // Invalid directory.
3460021         return (-1);
3460022     }
3460023     //
3460024     // Check if it is an open directory stream.
3460025     //
3460026     if (dp->fdn < 0)
3460027     {
3460028         //
3460029         // The stream is closed.
3460030         //
3460031         errset (EBADF); // Invalid directory.
3460032         return (-1);
3460033     }
3460034     //
```

```
3460035 // Close the file descriptor. If there is an error,  
3460036 // the 'errno' variable will be set by 'close()'.  
3460037 //  
3460038 return (close (dp->fdn));  
3460039 }
```

95.4.3 lib/dirent/opendir.c



Si veda la sezione [88.89](#).

```
3470001 #include <dirent.h>  
3470002 #include <fcntl.h>  
3470003 #include <stdio.h>  
3470004 #include <sys/types.h>  
3470005 #include <sys/stat.h>  
3470006 #include <unistd.h>  
3470007 #include <errno.h>  
3470008 #include <stddef.h>  
3470009 //-----  
3470010 DIR *  
3470011 opendir (const char *path)  
3470012 {  
3470013     int fdn;  
3470014     int d;  
3470015     DIR *dp;  
3470016     struct stat file_status;  
3470017     //  
3470018     // Function 'opendir()' is used only for reading.  
3470019     //  
3470020     fdn = open (path, O_RDONLY);  
3470021     //  
3470022     // Check the file descriptor returned.  
3470023     //  
3470024     if (fdn < 0)  
3470025     {  
3470026         //  
3470027         // The variable 'errno' is already set:
```

```
3470028         // EINVAL
3470029         // EMFILE
3470030         // ENFILE
3470031         //
3470032         errset (errno);
3470033         return (NULL);
3470034     }
3470035     //
3470036     // Set the 'FD_CLOEXEC' flag for that file
3470037     // descriptor.
3470038     //
3470039     if (fcntl (fdn, F_SETFD, FD_CLOEXEC) != 0)
3470040     {
3470041         //
3470042         // The variable 'errno' is already set:
3470043         // EBADF
3470044         //
3470045         errset (errno);
3470046         close (fdn);
3470047         return (NULL);
3470048     }
3470049     //
3470050     //
3470051     //
3470052     if (fstat (fdn, &file_status) != 0)
3470053     {
3470054         //
3470055         // Error should be already set.
3470056         //
3470057         errset (errno);
3470058         close (fdn);
3470059         return (NULL);
3470060     }
3470061     //
3470062     // Verify it is a directory
3470063     //
3470064     if (!S_ISDIR (file_status.st_mode))
```

```
3470065     {
3470066         //
3470067         // It is not a directory!
3470068         //
3470069         close (fdn);
3470070         errset (ENOTDIR); // Is not a directory.
3470071         return (NULL);
3470072     }
3470073     //
3470074     // A valid file descriptor is available: must find a
3470075     // free
3470076     // '_directory_stream[]' slot.
3470077     //
3470078     for (d = 0; d < DOPEN_MAX; d++)
3470079     {
3470080         if (_directory_stream[d].fdn < 0)
3470081         {
3470082             //
3470083             // Found a free slot: set it up.
3470084             //
3470085             dp = &(_directory_stream[d]);
3470086             dp->fdn = fdn;
3470087             //
3470088             // Return the directory pointer.
3470089             //
3470090             return (dp);
3470091         }
3470092     }
3470093     //
3470094     // If we are here, there was no free directory slot
3470095     // available.
3470096     //
3470097     close (fdn);
3470098     errset (EMFILE); // Too many file open.
3470099     return (NULL);
3470100 }
```

95.4.4 lib/dirent/readdir.c



Si veda la sezione [88.98](#).

```
3480001 #include <dirent.h>
3480002 #include <fcntl.h>
3480003 #include <sys/types.h>
3480004 #include <sys/stat.h>
3480005 #include <unistd.h>
3480006 #include <errno.h>
3480007 #include <stddef.h>
3480008 //-----
3480009 struct dirent *
3480010 readdir (DIR * dp)
3480011 {
3480012     ssize_t size;
3480013     //
3480014     // Check for a valid argument.
3480015     //
3480016     if (dp == NULL)
3480017     {
3480018         //
3480019         // Not a valid pointer.
3480020         //
3480021         errset (EBADF); // Invalid directory.
3480022         return (NULL);
3480023     }
3480024     //
3480025     // Check if it is an open directory stream.
3480026     //
3480027     if (dp->fdn < 0)
3480028     {
3480029         //
3480030         // The stream is closed.
3480031         //
3480032         errset (EBADF); // Invalid directory.
3480033         return (NULL);
3480034     }
```



```
3480035 //
3480036 // Read the directory.
3480037 //
3480038 size = read (dp->fdn, &(dp->dir), (size_t) 16);
3480039 //
3480040 // Fix the null termination, if the name is very
3480041 // long.
3480042 //
3480043 dp->dir.d_name[NAME_MAX] = '\0';
3480044 //
3480045 // Check what was read.
3480046 //
3480047 if (size == 0)
3480048     {
3480049         //
3480050         // End of directory, but it is not an error.
3480051         //
3480052         return (NULL);
3480053     }
3480054 //
3480055 if (size < 0)
3480056     {
3480057         //
3480058         // This is an error. The variable 'errno' is
3480059         // already set.
3480060         //
3480061         errset (errno);
3480062         return (NULL);
3480063     }
3480064 //
3480065 if (dp->dir.d_ino == 0)
3480066     {
3480067         //
3480068         // This is a null directory record.
3480069         // Should try to read the next one.
3480070         //
3480071         return (readdir (dp));
```

```
3480072     }
3480073     //
3480074     if (strlen (dp->dir.d_name) == 0)
3480075     {
3480076         //
3480077         // This is a bad directory record: try to read
3480078         // next.
3480079         //
3480080         return (readdir (dp));
3480081     }
3480082     //
3480083     // A valid directory record should be available now.
3480084     //
3480085     return (&(dp->dir));
3480086 }
```

95.4.5 lib/dirent/rewinddir.c



Si veda la sezione [88.101](#).

```
3490001 #include <dirent.h>
3490002 #include <fcntl.h>
3490003 #include <sys/types.h>
3490004 #include <sys/stat.h>
3490005 #include <unistd.h>
3490006 #include <errno.h>
3490007 #include <stddef.h>
3490008 #include <stdio.h>
3490009 //-----
3490010 void
3490011 rewinddir (DIR * dp)
3490012 {
3490013     FILE *fp;
3490014     //
3490015     // Check for a valid argument.
3490016     //
3490017     if (dp == NULL)
```

```
3490018     {
3490019         //
3490020         // Nothing to rewind, and no error to set.
3490021         //
3490022         return;
3490023     }
3490024     //
3490025     // Check if it is an open directory stream.
3490026     //
3490027     if (dp->fdn < 0)
3490028     {
3490029         //
3490030         // The stream is closed.
3490031         // Nothing to rewind, and no error to set.
3490032         //
3490033         return;
3490034     }
3490035     //
3490036     //
3490037     //
3490038     fp = &_stream[dp->fdn];
3490039     //
3490040     rewind (fp);
3490041 }
```

95.5 os32: «lib/errno.h»

Si veda la sezione [88.20](#).

```
3500001 #ifndef _ERRNO_H
3500002 #define _ERRNO_H          1
3500003 //-----
3500004 #include <limits.h>
3500005 #include <string.h>
3500006 #include <sys/os32.h>
3500007 #include <kernel/lib_k.h>
3500008
```

```
3500009 //-----
3500010 // The variable 'errno' is standard, but 'errln' and
3500011 // 'errfn' are added to keep track of the error source.
3500012 // Variable 'errln' is used to save the source file
3500013 // line number; variable 'errfn' is used to save the
3500014 // source file name. To set these variable in a
3500015 // consistent way it is also added a macroinstruction:
3500016 // 'errset'.
3500017 //-----
3500018 extern int errno;
3500019 extern int errln;
3500020 extern char errfn[PATH_MAX];
3500021 //
3500022 #define errset(e) \
3500023     (errln = __LINE__, \
3500024      strncpy (errfn, __FILE__, PATH_MAX), \
3500025      errno = e)
3500026 //-----
3500027 // Standard POSIX 'errno' macro variables.
3500028 //-----
3500029 #define E2BIG          1      // Argument list too
3500030                          // long.
3500031 #define EACCES        2      // Permission denied.
3500032 #define EADDRINUSE    3      // Address in use.
3500033 #define EADDRNOTAVAIL 4      // Address not
3500034                          // available.
3500035 #define EAFNOSUPPORT  5      // Address family not
3500036                          // supported.
3500037 #define EAGAIN        6      // Resource
3500038                          // unavailable, try
3500039                          // again.
3500040 #define EALREADY      7      // Connection already
3500041                          // in progress.
3500042 #define EBADF         8      // Bad file
3500043                          // descriptor.
3500044 #define EBADMSG       9      // Bad message.
3500045 #define EBUSY        10     // Device or resource
```

```
3500046 // busy.
3500047 #define ECANCELED 11 // Operation canceled.
3500048 #define ECHILD 12 // No child processes.
3500049 #define ECONNABORTED 13 // Connection aborted.
3500050 #define ECONNREFUSED 14 // Connection refused.
3500051 #define ECONNRESET 15 // Connection reset.
3500052 #define EDEADLK 16 // Resource deadlock
3500053 // would occur.
3500054 #define EDESTADDRREQ 17 // Destination address
3500055 // required.
3500056 #define EDOM 18 // Mathematics
3500057 // argument out of
3500058 // domain of
3500059 // function.
3500060 #define EDQUOT 19 // Reserved.
3500061 #define EEXIST 20 // File exists.
3500062 #define EFAULT 21 // Bad address.
3500063 #define EFBIG 22 // File too large.
3500064 #define EHOSTUNREACH 23 // Host is
3500065 // unreachable.
3500066 #define EIDRM 24 // Identifier removed.
3500067 #define EILSEQ 25 // Illegal byte
3500068 // sequence.
3500069 #define EINPROGRESS 26 // Operation in
3500070 // progress.
3500071 #define EINTR 27 // Interrupted
3500072 // function.
3500073 #define EINVAL 28 // Invalid argument.
3500074 #define EIO 29 // I/O error.
3500075 #define EISCONN 30 // Socket is
3500076 // connected.
3500077 #define EISDIR 31 // Is a directory.
3500078 #define ELOOP 32 // Too many levels of
3500079 // symbolic links.
3500080 #define EMFILE 33 // Too many open
3500081 // files.
3500082 #define EMLINK 34 // Too many links.
```

```
3500083 #define EMSGSIZE 35 // Message too large.
3500084 #define EMULTIHOP 36 // Reserved.
3500085 #define ENAMETOOLONG 37 // Filename too long.
3500086 #define ENETDOWN 38 // Network is down.
3500087 #define ENETRESET 39 // Connection aborted
3500088 // by network.
3500089 #define ENETUNREACH 40 // Network
3500090 // unreachable.
3500091 #define ENFILE 41 // Too many files open
3500092 // in system.
3500093 #define ENOBUFS 42 // No buffer space
3500094 // available.
3500095 #define ENODATA 43 // No message is
3500096 // available on the
3500097 // stream head
3500098 // read queue.
3500099 #define ENODEV 44 // No such device.
3500100 #define ENOENT 45 // No such file or
3500101 // directory.
3500102 #define ENOEXEC 46 // Executable file
3500103 // format error.
3500104 #define ENOLCK 47 // No locks available.
3500105 #define ENOLINK 48 // Reserved.
3500106 #define ENOMEM 49 // Not enough space.
3500107 #define ENOMSG 50 // No message of the
3500108 // desired type.
3500109 #define ENOPROTOPT 51 // Protocol not
3500110 // available.
3500111 #define ENOSPC 52 // No space left on
3500112 // device.
3500113 #define ENOSR 53 // No stream
3500114 // resources.
3500115 #define ENOSTR 54 // Not a stream.
3500116 #define ENOSYS 55 // Function not
3500117 // supported.
3500118 #define ENOTCONN 56 // The socket is not
3500119 // connected.
```

```
3500120 #define ENOTDIR 57 // Not a directory.
3500121 #define ENOTEMPTY 58 // Directory not
3500122 // empty.
3500123 #define ENOTSOCK 59 // Not a socket.
3500124 #define ENOTSUP 60 // Not supported.
3500125 #define ENOTTY 61 // Inappropriate I/O
3500126 // control operation.
3500127 #define ENXIO 62 // No such device or
3500128 // address.
3500129 #define EOPNOTSUPP 63 // Operation not
3500130 // supported on
3500131 // socket.
3500132 #define EOVERFLOW 64 // Value too large to
3500133 // be stored in data
3500134 // type.
3500135 #define EPERM 65 // Operation not
3500136 // permitted.
3500137 #define EPIPE 66 // Broken pipe.
3500138 #define EPROTO 67 // Protocol error.
3500139 #define EPROTONOSUPPORT 68 // Protocol not
3500140 // supported.
3500141 #define EPROTOTYPE 69 // Protocol wrong type
3500142 // for socket.
3500143 #define ERANGE 70 // Result too large.
3500144 #define EROFS 71 // Read-only file
3500145 // system.
3500146 #define ESPIPE 72 // Invalid seek.
3500147 #define ESRCH 73 // No such process.
3500148 #define ESTALE 74 // Reserved.
3500149 #define ETIME 75 // Stream ioctl()
3500150 // timeout.
3500151 #define ETIMEDOUT 76 // Connection timed
3500152 // out.
3500153 #define ETXTBSY 77 // Text file busy.
3500154 #define EWOULDBLOCK 78 // Operation would
3500155 // block (may be the
3500156 // same as EAGAIN).
```

```
3500157 #define EXDEV          79          // Cross-device link.
3500158 //-----
3500159 // Added os32 errors.
3500160 //-----
3500161 #define EUNKNOWN      (-1)       // Unknown
3500162                                     // error.
3500163 #define E_NO_MEDIUM   80         // No medium
3500164                                     // found.
3500165 #define E_MEDIUM      81         // Medium
3500166                                     // reported
3500167                                     // error.
3500168 #define E_FILE_TYPE   82         // File type
3500169                                     // not
3500170                                     // compatible.
3500171 #define E_ROOT_INODE_NOT_CACHED 83 // The root
3500172                                     // directory
3500173                                     // inode is
3500174                                     // not cached.
3500175 #define E_CANNOT_READ_SUPERBLOCK 84 // Cannot read
3500176                                     // super
3500177                                     // block.
3500178 #define E_MAP_INODE_TOO_BIG      85 // Map inode
3500179                                     // too big.
3500180 #define E_MAP_ZONE_TOO_BIG       86 // Map zone
3500181                                     // too big.
3500182 #define E_DATA_ZONE_TOO_BIG      87 // Data zone
3500183                                     // too big.
3500184 #define E_CANNOT_FIND_ROOT_DEVICE 88 // Cannot find
3500185                                     // root
3500186                                     // device.
3500187 #define E_CANNOT_FIND_ROOT_INODE 89 // Cannot find
3500188                                     // root inode.
3500189 #define E_FILE_TYPE_UNSUPPORTED  90 // File type
3500190                                     // unsupported.
3500191 #define E_ENV_TOO_BIG            91 // Environment
3500192                                     // too big.
3500193 #define E_LIMIT                  92 // Exceeded
```



```

3500194                                     // implementa-
3500195                                     // tion limits.
3500196 #define E_NOT_MOUNTED                    93 // Not
3500197                                     // mounted.
3500198 #define E_NOT_IMPLEMENTED                94 // Not
3500199                                     // implemented.
3500200 #define E_HARDWARE_FAULT                 95 // Hardware
3500201                                     // fault.
3500202 #define E_DRIVER_FAULT                   96 // Driver
3500203                                     // fault.
3500204 #define E_PIPE_FULL                      97 // Pipe full.
3500205 #define E_PIPE_EMPTY                    98 // Pipe empty.
3500206 #define E_PART_TYPE_NOT_MINIX           99 // Not a Minix
3500207                                     // partition
3500208                                     // type.
3500209 #define E_FS_TYPE_NOT_SUPPORTED          100 // File system
3500210                                     // type not
3500211                                     // supported.
3500212 #define E_PDU_TOO_BIG                    101 // PDU too
3500213                                     // big.
3500214 #define E_ARP_MISSING                    102 // ARP missing
3500215                                     // address.
3500216 //-----
3500217 // Default descriptions for errors.
3500218 //-----
3500219 #define TEXT_E2BIG                        "Argument list too long."
3500220 #define TEXT_EACCES                       "Permission denied."
3500221 #define TEXT_EADDRINUSE                   "Address in use."
3500222 #define TEXT_EADDRNOTAVAIL               "Address not available."
3500223 #define TEXT_EAFNOSUPPORT                 "Address family not " \
3500224                                     "supported."
3500225 #define TEXT_EAGAIN                       "Resource unavailable, " \
3500226                                     "try again."
3500227 #define TEXT_EALREADY                     "Connection already in " \
3500228                                     "progress."
3500229 #define TEXT_EBADF                        "Bad file descriptor."
3500230 #define TEXT_EBADMSG                      "Bad message."

```

```
3500231 #define TEXT_EBUSY "Device or resource busy."
3500232 #define TEXT_ECANCELED "Operation canceled."
3500233 #define TEXT_ECHILD "No child processes."
3500234 #define TEXT_ECONNABORTED "Connection aborted."
3500235 #define TEXT_ECONNREFUSED "Connection refused."
3500236 #define TEXT_ECONNRESET "Connection reset."
3500237 #define TEXT_EDEADLK "Resource deadlock " \
3500238 "would occur."
3500239 #define TEXT_EDESTADDRREQ "Destination address " \
3500240 "required."
3500241 #define TEXT_EDOM "Mathematics argument " \
3500242 "out of " \
3500243 "domain of function."
3500244 #define TEXT_EDQUOT "Reserved error: EDQUOT"
3500245 #define TEXT_EEXIST "File exists."
3500246 #define TEXT_EFAULT "Bad address."
3500247 #define TEXT_EFBIG "File too large."
3500248 #define TEXT_EHOSTUNREACH "Host is unreachable."
3500249 #define TEXT_EIDRM "Identifier removed."
3500250 #define TEXT_EILSEQ "Illegal byte sequence."
3500251 #define TEXT_EINPROGRESS "Operation in progress."
3500252 #define TEXT_EINTR "Interrupted function."
3500253 #define TEXT_EINVAL "Invalid argument."
3500254 #define TEXT_EIO "I/O error."
3500255 #define TEXT_EISCONN "Socket is connected."
3500256 #define TEXT_EISDIR "Is a directory."
3500257 #define TEXT_ELOOP "Too many levels of " \
3500258 "symbolic links."
3500259 #define TEXT_EMFILE "Too many open files."
3500260 #define TEXT_EMLINK "Too many links."
3500261 #define TEXT EMSGSIZE "Message too large."
3500262 #define TEXT_EMULTIHOP "Reserved error: " \
3500263 "EMULTIHOP"
3500264 #define TEXT_ENAMETOOLONG "Filename too long."
3500265 #define TEXT_ENETDOWN "Network is down."
3500266 #define TEXT_ENETRESET "Connection aborted by " \
3500267 "network."
```

```
3500268 #define TEXT_ENETUNREACH "Network unreachable."
3500269 #define TEXT_ENFILE "Too many files open " \
3500270 "in system."
3500271 #define TEXT_ENOBUFS "No buffer space " \
3500272 "available."
3500273 #define TEXT_ENODATA "No message is " \
3500274 "available on the " \
3500275 "stream head read queue."
3500276 #define TEXT_ENODEV "No such device."
3500277 #define TEXT_ENOENT "No such file or " \
3500278 "directory."
3500279 #define TEXT_ENOEXEC "Executable file " \
3500280 "format error."
3500281 #define TEXT_ENOLCK "No locks available."
3500282 #define TEXT_ENOLINK "Reserved error: ENOLINK"
3500283 #define TEXT_ENOMEM "Not enough space."
3500284 #define TEXT_ENOMSG "No message of the " \
3500285 "desired type."
3500286 #define TEXT_ENOPROTOOPT "Protocol not available."
3500287 #define TEXT_ENOSPC "No space left on device."
3500288 #define TEXT_ENOSR "No stream resources."
3500289 #define TEXT_ENOSTR "Not a stream."
3500290 #define TEXT_ENOSYS "Function not supported."
3500291 #define TEXT_ENOTCONN "The socket is not " \
3500292 "connected."
3500293 #define TEXT_ENOTDIR "Not a directory."
3500294 #define TEXT_ENOTEMPTY "Directory not empty."
3500295 #define TEXT_ENOTSOCK "Not a socket."
3500296 #define TEXT_ENOTSUP "Not supported."
3500297 #define TEXT_ENOTTY "Inappropriate I/O " \
3500298 "control operation."
3500299 #define TEXT_ENXIO "No such device or " \
3500300 "address."
3500301 #define TEXT_EOPNOTSUPP "Operation not " \
3500302 "supported on socket."
3500303 #define TEXT_EOVERFLOW "Value too large to be " \
3500304 "stored in data type."
```

```
3500305 #define TEXT_EPERM "Operation not permitted."
3500306 #define TEXT_EPIPE "Broken pipe."
3500307 #define TEXT_EPROTO "Protocol error."
3500308 #define TEXT_EPROTONOSUPPORT "Protocol not supported."
3500309 #define TEXT_EPROTOTYPE "Protocol wrong type " \
3500310 "for socket."
3500311 #define TEXT_ERANGE "Result too large."
3500312 #define TEXT_EROFS "Read-only file system."
3500313 #define TEXT_ESPIPE "Invalid seek."
3500314 #define TEXT_ESRCH "No such process."
3500315 #define TEXT_ESTALE "Reserved error: ESTALE"
3500316 #define TEXT_ETIME "Stream ioctl() timeout."
3500317 #define TEXT_ETIMEDOUT "Connection timed out."
3500318 #define TEXT_ETXTBSY "Text file busy."
3500319 #define TEXT_EWOULDBLOCK "Operation would block."
3500320 #define TEXT_EXDEV "Cross-device link."
3500321 //-----
3500322 #define TEXT_EUNKNOWN \
3500323 "Unknown error."
3500324 #define TEXT_E_NO_MEDIUM \
3500325 "No medium found."
3500326 #define TEXT_E_MEDIUM \
3500327 "Medium reported error"
3500328 #define TEXT_E_FILE_TYPE \
3500329 "File type not compatible."
3500330 #define TEXT_E_ROOT_INODE_NOT_CACHED \
3500331 "The root directory inode is not cached."
3500332 #define TEXT_E_CANNOT_READ_SUPERBLOCK \
3500333 "Cannot read super block."
3500334 #define TEXT_E_MAP_INODE_TOO_BIG \
3500335 "Map inode too big."
3500336 #define TEXT_E_MAP_ZONE_TOO_BIG \
3500337 "Map zone too big."
3500338 #define TEXT_E_DATA_ZONE_TOO_BIG \
3500339 "Data zone too big."
3500340 #define TEXT_E_CANNOT_FIND_ROOT_DEVICE \
3500341 "Cannot find root device."
```

```

3500342 #define TEXT_E_CANNOT_FIND_ROOT_INODE \
3500343     "Cannot find root inode."
3500344 #define TEXT_E_FILE_TYPE_UNSUPPORTED \
3500345     "File type unsupported."
3500346 #define TEXT_E_ENV_TOO_BIG \
3500347     "Environment too big."
3500348 #define TEXT_E_LIMIT \
3500349     "Exceeded implementation limits."
3500350 #define TEXT_E_NOT_MOUNTED \
3500351     "Not mounted."
3500352 #define TEXT_E_NOT_IMPLEMENTED \
3500353     "Not implemented."
3500354 #define TEXT_E_HARDWARE_FAULT \
3500355     "Hardware fault."
3500356 #define TEXT_E_DRIVER_FAULT \
3500357     "Driver fault."
3500358 #define TEXT_E_PIPE_FULL \
3500359     "Pipe full."
3500360 #define TEXT_E_PIPE_EMPTY \
3500361     "Pipe empty."
3500362 #define TEXT_E_PART_TYPE_NOT_MINIX \
3500363     "Not a Minix partition type."
3500364 #define TEXT_E_FS_TYPE_NOT_SUPPORTED \
3500365     "File system type not supported."
3500366 #define TEXT_E_PDU_TOO_BIG \
3500367     "PDU too big."
3500368 #define TEXT_E_ARP_MISSING \
3500369     "ARP missing address."
3500370 //-----
3500371 #endif

```

95.5.1 lib/errno/errno.c 1846

95.5.1 lib/errno/errno.c



Si veda la sezione 88.20.

```
3510001 //-----
3510002 // This file does not include the 'errno.h' header,
3510003 // because here 'errno' should not be declared as an
3510004 // extern variable!
3510005 //-----
3510006 #include <limits.h>
3510007 //-----
3510008 // The variable 'errno' is standard, but 'errln' and
3510009 // 'errfn' are added to keep track of the error source.
3510010 // Variable 'errln' is used to save the source file
3510011 // line number; variable 'errfn' is used to save the
3510012 // source file name.
3510013 // To set these variable in a consistent way it is
3510014 // also added a macroinstruction: 'errset'.
3510015 //-----
3510016 int errno;
3510017 int errln;
3510018 char errfn[PATH_MAX];
3510019 //-----
```

95.6 os32: «lib/fcntl.h»



Si veda la sezione 91.3.

```
3520001 #ifndef _FCNTL_H
3520002 #define _FCNTL_H          1
3520003
3520004 #include <sys/types.h> // mode_t
3520005                       // off_t
3520006                       // pid_t
3520007 //-----
3520008 // Values for the second parameter of function
3520009 // 'fcntl()'.
3520010 //-----
```

```
3520011 #define F_DUPFD 0 // Duplicate file
3520012 // descriptor.
3520013 #define F_GETFD 1 // Get file descriptor
3520014 // flags.
3520015 #define F_SETFD 2 // Set file descriptor
3520016 // flags.
3520017 #define F_GETFL 3 // Get file status
3520018 // flags.
3520019 #define F_SETFL 4 // Set file status
3520020 // flags.
3520021 #define F_GETLK 5 // Get record locking
3520022 // information.
3520023 #define F_SETLK 6 // Set record locking
3520024 // information.
3520025 #define F_SETLKW 7 // Set record locking
3520026 // information;
3520027 // wait if blocked.
3520028 #define F_GETOWN 8 // Set owner of
3520029 // socket.
3520030 #define F_SETOWN 9 // Get owner of
3520031 // socket.
3520032 //-----
3520033 // Flags to be set with:
3520034 // fcntl (fd, F_SETFD, ...);
3520035 //-----
3520036 #define FD_CLOEXEC 1 // Close the file
3520037 // descriptor upon
3520038 // execution of an
3520039 // exec() family
3520040 // function.
3520041 //-----
3520042 // Values for type 'l_type', used for record locking
3520043 // with 'fcntl()'.
3520044 //-----
3520045 #define F_RDLCK 0 // Read lock.
3520046 #define F_WRLCK 1 // Write lock.
3520047 #define F_UNLCK 2 // Remove lock.
```

```
3520048 //-----
3520049 // Flags for file creation, in place of 'oflag'
3520050 // parameter for function 'open()'.
3520051 //-----
3520052 #define O_CREAT          000010 // Create file if it
3520053 // does not exist.
3520054 #define O_EXCL          000020 // Exclusive use flag.
3520055 #define O_NOCTTY        000040 // Do not assign a
3520056 // controlling
3520057 // terminal.
3520058 #define O_TRUNC         000100 // Truncation flag.
3520059 //-----
3520060 // Flags for the file status, used with 'open()' and
3520061 // 'fcntl()'.
3520062 //-----
3520063 #define O_APPEND        000200 // Write append.
3520064 #define O_DSYNC         000400 // Synchronized write
3520065 // operations.
3520066 #define O_NONBLOCK      001000 // Non-blocking mode.
3520067 #define O_RSYNC         002000 // Synchronized read
3520068 // operations.
3520069 #define O_SYNC          004000 // Synchronized read
3520070 // and write.
3520071 //-----
3520072 // File access mask selection.
3520073 //-----
3520074 #define O_ACCMODE       000003 // Mask to select the
3520075 // last three bits,
3520076 // used to specify the
3520077 // main access
3520078 // modes: read, write
3520079 // and both.
3520080 //-----
3520081 // Main access modes.
3520082 //-----
3520083 #define O_RDONLY        000001 // Read.
3520084 #define O_WRONLY        000002 // Write.
```



```

3520085 #define O_RDWR          (O_RDONLY | O_WRONLY)    // [1]
3520086 //
3520087 // [1] Both read and write.
3520088 //
3520089 //-----
3520090 // Structure 'flock', used to file lock for POSIX
3520091 // standard. It is not used inside os32.
3520092 //-----
3520093 struct flock
3520094 {
3520095     short int l_type;      // Type of lock: F_RDLCK,
3520096     // F_WRLCK, or F_UNLCK.
3520097     short int l_whence;   // Start reference point.
3520098     off_t l_start;       // Offset, from 'l_whence',
3520099     // for the area start.
3520100     off_t l_len;        // Locked area size. Zero means up to
3520101     // the end of the file.
3520102     pid_t l_pid;        // The process id blocking the area.
3520103 };
3520104 //-----
3520105 // Function prototypes.
3520106 //-----
3520107 int creat (const char *path, mode_t mode);
3520108 int fcntl (int fdn, int cmd, ...);
3520109 int open (const char *path, int oflags, ...);
3520110 //-----
3520111
3520112 #endif

```

95.6.1	lib/fcntl/creat.c	1850
95.6.2	lib/fcntl/fcntl.c	1850
95.6.3	lib/fcntl/open.c	1852

95.6.1 lib/fcntl/creat.c



Si veda la sezione [88.14](#).

```
3530001 #include <fcntl.h>
3530002 #include <sys/types.h>
3530003 //-----
3530004 int
3530005 creat (const char *path, mode_t mode)
3530006 {
3530007     return (open (path, O_WRONLY | O_CREAT | O_TRUNC, mode));
3530008 }
```

95.6.2 lib/fcntl/fcntl.c



Si veda la sezione [87.18](#).

```
3540001 #include <fcntl.h>
3540002 #include <stdarg.h>
3540003 #include <stddef.h>
3540004 #include <string.h>
3540005 #include <errno.h>
3540006 #include <sys/os32.h>
3540007 #include <limits.h>
3540008 //-----
3540009 int
3540010 fcntl (int fdn, int cmd, ...)
3540011 {
3540012     va_list ap;
3540013     sysmsg_fcntl_t msg;
3540014     va_start (ap, cmd);
3540015     //
3540016     // Well known arguments.
3540017     //
3540018     msg.fdn = fdn;
3540019     msg.cmd = cmd;
3540020     //
3540021     // Select other arguments.
```

```
3540022 //
3540023 switch (cmd)
3540024 {
3540025     case F_DUPFD:
3540026     case F_SETFD:
3540027     case F_SETFL:
3540028         msg.arg = va_arg (ap, int);
3540029         break;
3540030     case F_GETFD:
3540031     case F_GETFL:
3540032         break;
3540033     case F_GETOWN:
3540034     case F_SETOWN:
3540035     case F_GETLK:
3540036     case F_SETLK:
3540037     case F_SETLKW:
3540038         errset (E_NOT_IMPLEMENTED);           // Not
3540039         // implemented.
3540040         return (-1);
3540041     default:
3540042         errset (EINVAL); // Not implemented.
3540043         return (-1);
3540044 }
3540045 //
3540046 // Do the system call.
3540047 //
3540048 sys (SYS_FCNTL, &msg, (sizeof msg));
3540049 errno = msg.errno;
3540050 errln = msg.errln;
3540051 strncpy (errfn, msg.errfn, PATH_MAX);
3540052 return (msg.ret);
3540053 }
```

95.6.3 lib/fcntl/open.c



Si veda la sezione [87.37](#).

```
3550001 #include <fcntl.h>
3550002 #include <stdarg.h>
3550003 #include <stddef.h>
3550004 #include <string.h>
3550005 #include <errno.h>
3550006 #include <sys/os32.h>
3550007 #include <limits.h>
3550008 //-----
3550009 int
3550010 open (const char *path, int oflags, ...)
3550011 {
3550012     va_list ap;
3550013     sysmsg_open_t msg;
3550014     va_start (ap, oflags);
3550015     msg.path = path;
3550016     msg.flags = oflags;
3550017     msg.mode = va_arg (ap, mode_t);
3550018     sys (SYS_OPEN, &msg, (sizeof msg));
3550019     errno = msg.errno;
3550020     errln = msg.errln;
3550021     strncpy (errfn, msg.errfn, PATH_MAX);
3550022     return (msg.ret);
3550023 }
```

95.7 os32: «lib/grp.h»



Si veda la sezione [91.3](#).

```
3560001 #ifndef _GRP_H
3560002 #define _GRP_H          1
3560003 //-----
3560004 #include <restrict.h>
3560005 #include <sys/types.h> // gid_t, uid_t
3560006 //-----
```

```

3560007 #define GR_MEM_MAX 32
3560008 struct group
3560009 {
3560010     char *gr_name;
3560011     char *gr_passwd;
3560012     gid_t gr_gid;
3560013     char *gr_mem[GR_MEM_MAX];
3560014 };
3560015 //-----
3560016 struct group *getgrent (void);
3560017 void setgrent (void);
3560018 void endgrent (void);
3560019 struct group *getgrnam (const char *name);
3560020 struct group *getgrgid (gid_t gid);
3560021 //-----
3560022 #endif

```

95.7.1 lib/grp/grent.c 1853

95.7.1 lib/grp/grent.c

Si veda la sezione [88.53](#).

```

3570001 #include <grp.h>
3570002 #include <stdio.h>
3570003 #include <string.h>
3570004 #include <stdlib.h>
3570005 //-----
3570006 static char buffer[BUFSIZ];
3570007 static struct group gr;
3570008 static FILE *fp = NULL;
3570009 //-----
3570010 struct group *
3570011 getgrent (void)
3570012 {
3570013     void *pstatus;
3570014     char *char_gid;

```



```
3570015     int i;
3570016     //
3570017     if (fp == NULL)
3570018     {
3570019         fp = fopen ("/etc/group", "r");
3570020         if (fp == NULL)
3570021         {
3570022             return NULL;
3570023         }
3570024     }
3570025     //
3570026     pstatus = fgets (buffer, BUFSIZ, fp);
3570027     if (pstatus == NULL)
3570028     {
3570029         return (NULL);
3570030     }
3570031     //
3570032     // The parse is made with 'strtok()'. Please notice
3570033     // that
3570034     // 'strtok()' will not parse a line like the
3570035     // following:
3570036     // user::233:
3570037     // The password field *must* have something,
3570038     // otherwise the
3570039     // GID will take the password place.
3570040     // 'strtok()' will consider '::' the same as ':'!
3570041     //
3570042     gr.gr_name = strtok (buffer, ":");
3570043     gr.gr_passwd = strtok (NULL, ":");
3570044     char_gid = strtok (NULL, ":");
3570045     for (i = 0; i < GR_MEM_MAX; i++)
3570046     {
3570047         gr.gr_mem[i] = strtok (NULL, ",\n");
3570048     }
3570049     gr.gr_gid = (gid_t) atoi (char_gid);
3570050     //
3570051     return (&gr);
```

```
3570052 }
3570053
3570054 //-----
3570055 void
3570056 endgrent (void)
3570057 {
3570058     int status;
3570059     //
3570060     if (fp != NULL)
3570061     {
3570062         status = fclose (fp);
3570063         if (status != 0)
3570064         {
3570065             perror (NULL);
3570066             fp = NULL;
3570067         }
3570068     }
3570069 }
3570070
3570071 //-----
3570072 void
3570073 setgrent (void)
3570074 {
3570075     if (fp != NULL)
3570076     {
3570077         rewind (fp);
3570078     }
3570079 }
3570080
3570081 //-----
3570082 struct group *
3570083 getgrnam (const char *name)
3570084 {
3570085     struct group *gr;
3570086     //
3570087     setgrent ();
3570088     //
```

```
3570089     for (;;)
3570090     {
3570091         gr = getgrent ();
3570092         if (gr == NULL)
3570093             {
3570094                 return (NULL);
3570095             }
3570096         if (strcmp (gr->gr_name, name) == 0)
3570097             {
3570098                 return (gr);
3570099             }
3570100     }
3570101 }
3570102
3570103 //-----
3570104 struct group *
3570105 getgrgid (gid_t gid)
3570106 {
3570107     struct group *gr;
3570108     //
3570109     setgrent ();
3570110     //
3570111     for (;;)
3570112     {
3570113         gr = getgrent ();
3570114         if (gr == NULL)
3570115             {
3570116                 return (NULL);
3570117             }
3570118         if (gr->gr_gid == gid)
3570119             {
3570120                 return (gr);
3570121             }
3570122     }
3570123 }
```


95.8 os32: «lib/inttypes.h»



Si veda la sezione [91.3](#).

```
3580001 #ifndef _INTTYPES_H
3580002 #define _INTTYPES_H      1
3580003 //-----
3580004 #include <stdint.h>
3580005 #include <wchar_t.h>
3580006 #include <restrict.h>
3580007 //-----
3580008 typedef struct
3580009 {
3580010     intmax_t quot;
3580011     intmax_t rem;
3580012 } imaxdiv_t;
3580013 //
3580014 imaxdiv_t imaxdiv (intmax_t numer, intmax_t denom);
3580015 //-----
3580016 // Output typesetting.
3580017 //-----
3580018 #define PRId8           "d"
3580019 #define PRId16          "d"
3580020 #define PRId32          "d"
3580021 #define PRId64          "lld"
3580022 //
3580023 #define PRIdLEAST8     "d"
3580024 #define PRIdLEAST16    "d"
3580025 #define PRIdLEAST32    "d"
3580026 #define PRIdLEAST64    "lld"
3580027 //
3580028 #define PRIdFAST8      "d"
3580029 #define PRIdFAST16     "d"
3580030 #define PRIdFAST32     "d"
3580031 #define PRIdFAST64     "lld"
3580032 //
3580033 #define PRIdMAX        "lld"
3580034 #define PRIdPTR        "d"
```

```
3580035 //
3580036 #define PRIi8 "i"
3580037 #define PRIi16 "i"
3580038 #define PRIi32 "i"
3580039 #define PRIi64 "lli"
3580040 //
3580041 #define PRIiLEAST8 "i"
3580042 #define PRIiLEAST16 "i"
3580043 #define PRIiLEAST32 "i"
3580044 #define PRIiLEAST64 "lli"
3580045 //
3580046 #define PRIiFAST8 "i"
3580047 #define PRIiFAST16 "i"
3580048 #define PRIiFAST32 "i"
3580049 #define PRIiFAST64 "lli"
3580050 //
3580051 #define PRIiMAX "lli"
3580052 #define PRIiPTR "i"
3580053 //
3580054 #define PRIb8 "b" // PRIb... is not
3580055 // standard!
3580056 #define PRIb16 "b" //
3580057 #define PRIb32 "b" //
3580058 #define PRIb64 "llb" //
3580059 // //
3580060 #define PRIbLEAST8 "b" //
3580061 #define PRIbLEAST16 "b" //
3580062 #define PRIbLEAST32 "b" //
3580063 #define PRIbLEAST64 "llb" //
3580064 // //
3580065 #define PRIbFAST8 "b" //
3580066 #define PRIbFAST16 "b" //
3580067 #define PRIbFAST32 "b" //
3580068 #define PRIbFAST64 "llb" //
3580069 // //
3580070 #define PRIbMAX "llb" //
3580071 #define PRIbPTR "b" //
```

```
3580072 //
3580073 #define PRIo8 "o"
3580074 #define PRIo16 "o"
3580075 #define PRIo32 "o"
3580076 #define PRIo64 "llo"
3580077 //
3580078 #define PRIoLEAST8 "o"
3580079 #define PRIoLEAST16 "o"
3580080 #define PRIoLEAST32 "o"
3580081 #define PRIoLEAST64 "llo"
3580082 //
3580083 #define PRIoFAST8 "o"
3580084 #define PRIoFAST16 "o"
3580085 #define PRIoFAST32 "o"
3580086 #define PRIoFAST64 "llo"
3580087 //
3580088 #define PRIoMAX "llo"
3580089 #define PRIoPTR "o"
3580090 //
3580091 #define PRIu8 "u"
3580092 #define PRIu16 "u"
3580093 #define PRIu32 "u"
3580094 #define PRIu64 "llu"
3580095 //
3580096 #define PRIuLEAST8 "u"
3580097 #define PRIuLEAST16 "u"
3580098 #define PRIuLEAST32 "u"
3580099 #define PRIuLEAST64 "llu"
3580100 //
3580101 #define PRIuFAST8 "u"
3580102 #define PRIuFAST16 "u"
3580103 #define PRIuFAST32 "u"
3580104 #define PRIuFAST64 "llu"
3580105 //
3580106 #define PRIuMAX "llu"
3580107 #define PRIuPTR "u"
3580108 //
```

```
3580109 #define PRIx8 "x"
3580110 #define PRIx16 "x"
3580111 #define PRIx32 "x"
3580112 #define PRIx64 "llx"
3580113 //
3580114 #define PRIxLEAST8 "x"
3580115 #define PRIxLEAST16 "x"
3580116 #define PRIxLEAST32 "x"
3580117 #define PRIxLEAST64 "llx"
3580118 //
3580119 #define PRIxFAST8 "x"
3580120 #define PRIxFAST16 "x"
3580121 #define PRIxFAST32 "x"
3580122 #define PRIxFAST64 "llx"
3580123 //
3580124 #define PRIxMAX "llx"
3580125 #define PRIxPTR "x"
3580126 //
3580127 #define PRIX8 "X"
3580128 #define PRIX16 "X"
3580129 #define PRIX32 "X"
3580130 #define PRIX64 "llX"
3580131 //
3580132 #define PRIXLEAST8 "X"
3580133 #define PRIXLEAST16 "X"
3580134 #define PRIXLEAST32 "X"
3580135 #define PRIXLEAST64 "llX"
3580136 //
3580137 #define PRIXFAST8 "X"
3580138 #define PRIXFAST16 "X"
3580139 #define PRIXFAST32 "X"
3580140 #define PRIXFAST64 "llX"
3580141 //
3580142 #define PRIXMAX "llX"
3580143 #define PRIXPTR "X"
3580144 //-----
3580145 // Input scan and evaluation.
```

```
3580146 //-----
3580147 #define SCNd8 "hhd"
3580148 #define SCNd16 "hd"
3580149 #define SCNd32 "d"
3580150 #define SCNd64 "lld"
3580151 //
3580152 #define SCNdLEAST8 "hhd"
3580153 #define SCNdLEAST16 "hd"
3580154 #define SCNdLEAST32 "d"
3580155 #define SCNdLEAST64 "lld"
3580156 //
3580157 #define SCNdFAST8 "hhd"
3580158 #define SCNdFAST16 "d"
3580159 #define SCNdFAST32 "d"
3580160 #define SCNdFAST64 "lld"
3580161 //
3580162 #define SCNdMAX "lld"
3580163 #define SCNdPTR "d"
3580164 //
3580165 #define SCNi8 "hhi"
3580166 #define SCNi16 "hi"
3580167 #define SCNi32 "i"
3580168 #define SCNi64 "lli"
3580169 //
3580170 #define SCNiLEAST8 "hhi"
3580171 #define SCNiLEAST16 "hi"
3580172 #define SCNiLEAST32 "i"
3580173 #define SCNiLEAST64 "lli"
3580174 //
3580175 #define SCNiFAST8 "hhi"
3580176 #define SCNiFAST16 "i"
3580177 #define SCNiFAST32 "i"
3580178 #define SCNiFAST64 "lli"
3580179 //
3580180 #define SCNiMAX "lli"
3580181 #define SCNiPTR "i"
3580182 //
```

```
3580183 #define SCNb8 "hnb" // SCNb... is not
3580184 // standard!
3580185 #define SCNb16 "hb" //
3580186 #define SCNb32 "b" //
3580187 #define SCNb64 "llb" //
3580188 // //
3580189 #define SCNbLEAST8 "hnb" //
3580190 #define SCNbLEAST16 "hb" //
3580191 #define SCNbLEAST32 "b" //
3580192 #define SCNbLEAST64 "llb" //
3580193 // //
3580194 #define SCNbFAST8 "hnb" //
3580195 #define SCNbFAST16 "b" //
3580196 #define SCNbFAST32 "b" //
3580197 #define SCNbFAST64 "llb" //
3580198 // //
3580199 #define SCNbMAX "llb" //
3580200 #define SCNbPTR "b" //
3580201 // //
3580202 #define SCNo8 "hho"
3580203 #define SCNo16 "ho"
3580204 #define SCNo32 "o"
3580205 #define SCNo64 "llo"
3580206 // //
3580207 #define SCNoLEAST8 "hho"
3580208 #define SCNoLEAST16 "ho"
3580209 #define SCNoLEAST32 "o"
3580210 #define SCNoLEAST64 "llo"
3580211 // //
3580212 #define SCNoFAST8 "hho"
3580213 #define SCNoFAST16 "o"
3580214 #define SCNoFAST32 "o"
3580215 #define SCNoFAST64 "llo"
3580216 // //
3580217 #define SCNoMAX "llo"
3580218 #define SCNoPTR "o"
3580219 // //
```

```
3580220 #define SCNu8 "hhu"
3580221 #define SCNu16 "hu"
3580222 #define SCNu32 "u"
3580223 #define SCNu64 "llu"
3580224 //
3580225 #define SCNuLEAST8 "hhu"
3580226 #define SCNuLEAST16 "hu"
3580227 #define SCNuLEAST32 "u"
3580228 #define SCNuLEAST64 "llu"
3580229 //
3580230 #define SCNuFAST8 "hhu"
3580231 #define SCNuFAST16 "u"
3580232 #define SCNuFAST32 "u"
3580233 #define SCNuFAST64 "llu"
3580234 //
3580235 #define SCNuMAX "llu"
3580236 #define SCNuPTR "u"
3580237 //
3580238 #define SCNx8 "hhx"
3580239 #define SCNx16 "hx"
3580240 #define SCNx32 "x"
3580241 #define SCNx64 "llx"
3580242 //
3580243 #define SCNxLEAST8 "hhx"
3580244 #define SCNxLEAST16 "hx"
3580245 #define SCNxLEAST32 "x"
3580246 #define SCNxLEAST64 "llx"
3580247 //
3580248 #define SCNxFAST8 "hhx"
3580249 #define SCNxFAST16 "x"
3580250 #define SCNxFAST32 "x"
3580251 #define SCNxFAST64 "llx"
3580252 //
3580253 #define SCNxMAX "llx"
3580254 #define SCNxPTR "x"
3580255 //-----
3580256 intmax_t imaxabs (intmax_t j);
```

```

3580257 intmax_t strtouimax (const char *restrict nptr,
3580258                     char **restrict endptr, int base);
3580259 uintmax_t strtouimax (const char *restrict nptr,
3580260                       char **restrict endptr, int base);
3580261 intmax_t wcstouimax (const wchar_t * restrict nptr,
3580262                     wchar_t ** restrict endptr, int base);
3580263 uintmax_t wcstouimax (const wchar_t * restrict nptr,
3580264                       wchar_t ** restrict endptr, int base);
3580265 //-----
3580266 #endif

```

[95.8.1 lib/inttypes/imaxabs.c](#) 1864

[95.8.2 lib/inttypes/imaxdiv.c](#) 1865

95.8.1 lib/inttypes/imaxabs.c

«

Si veda la sezione [88.3](#).

```

3590001 #include <inttypes.h>
3590002 //-----
3590003 intmax_t
3590004 imaxabs (intmax_t j)
3590005 {
3590006     if (j < 0)
3590007     {
3590008         return -j;
3590009     }
3590010     else
3590011     {
3590012         return j;
3590013     }
3590014 }

```


95.8.2 lib/inttypes/imaxdiv.c



Si veda la sezione [88.17](#).

```

3600001 #include <inttypes.h>
3600002 //-----
3600003 imaxdiv_t
3600004 imaxdiv (intmax_t numer, intmax_t denom)
3600005 {
3600006     imaxdiv_t d;
3600007     d.quot = numer / denom;
3600008     d.rem = numer % denom;
3600009     return d;
3600010 }
```

95.9 os32: «lib/libgen.h»



Si veda la sezione [91.3](#).

```

3610001 #ifndef _LIBGEN_H
3610002 #define _LIBGEN_H      1
3610003
3610004 //-----
3610005 char *basename (char *path);
3610006 char *dirname (char *path);
3610007 //-----
3610008
3610009 #endif
```

[95.9.1](#) lib/libgen/basename.c 1866

[95.9.2](#) lib/libgen/dirname.c 1867

95.9.1 lib/libgen/basename.c

<<

Si veda la sezione 88.10.

```
3620001 #include <libgen.h>
3620002 #include <limits.h>
3620003 #include <stddef.h>
3620004 #include <string.h>
3620005 //-----
3620006 char *
3620007 basename (char *path)
3620008 {
3620009     static char *point = ".";    // When 'path' is
3620010     // NULL.
3620011     char *p;    // Pointer inside 'path'.
3620012     int i;    // Scan index inside 'path'.
3620013     //
3620014     // Empty path.
3620015     //
3620016     if (path == NULL || strlen (path) == 0)
3620017     {
3620018         return (point);
3620019     }
3620020     //
3620021     // Remove all final '/' if it exists, excluded the
3620022     // first character:
3620023     // 'i' is kept greater than zero.
3620024     //
3620025     for (i = (strlen (path) - 1);
3620026          i > 0 && path[i] == '/'; i--)
3620027     {
3620028         path[i] = 0;
3620029     }
3620030     //
3620031     // After removal of extra final '/', if there is
3620032     // only one '/', this
3620033     // is to be returned.
3620034     //
```

```

3620035     if (strncmp (path, "/", PATH_MAX) == 0)
3620036     {
3620037         return (path);
3620038     }
3620039     //
3620040     // If there are no '/'.
3620041     //
3620042     if (strchr (path, '/') == NULL)
3620043     {
3620044         return (path);
3620045     }
3620046     //
3620047     // Find the last '/' and calculate a pointer to the
3620048     // base name.
3620049     //
3620050     p = strrchr (path, (unsigned int) '/');
3620051     p++;
3620052     //
3620053     // Return the pointer to the base name.
3620054     //
3620055     return (p);
3620056 }

```

95.9.2 lib/libgen/dirname.c

Si veda la sezione [88.10](#).

```

3630001 #include <libgen.h>
3630002 #include <limits.h>
3630003 #include <stddef.h>
3630004 #include <string.h>
3630005 //-----
3630006 char *
3630007 dirname (char *path)
3630008 {
3630009     static char *point = ".";    // When 'path' is
3630010     // NULL.

```

```
3630011 char *p;          // Pointer inside 'path'.
3630012 int i;          // Scan index inside 'path'.
3630013 //
3630014 // Empty path.
3630015 //
3630016 if (path == NULL || strlen (path) == 0)
3630017 {
3630018     return (point);
3630019 }
3630020 //
3630021 // Simple cases.
3630022 //
3630023 if (strncmp (path, "/", PATH_MAX) == 0 ||
3630024     strncmp (path, ".", PATH_MAX) == 0 ||
3630025     strncmp (path, "..", PATH_MAX) == 0)
3630026 {
3630027     return (path);
3630028 }
3630029 //
3630030 // Remove all final '/' if it exists, excluded the
3630031 // first character:
3630032 // 'i' is kept greater than zero.
3630033 //
3630034 for (i = (strlen (path) - 1);
3630035     i > 0 && path[i] == '/'; i--)
3630036 {
3630037     path[i] = 0;
3630038 }
3630039 //
3630040 // After removal of extra final '/', if there is
3630041 // only one '/', this
3630042 // is to be returned.
3630043 //
3630044 if (strncmp (path, "/", PATH_MAX) == 0)
3630045 {
3630046     return (path);
3630047 }
```

```
3630048 //
3630049 // If there are no '/'
3630050 //
3630051 if (strchr (path, '/') == NULL)
3630052 {
3630053     return (point);
3630054 }
3630055 //
3630056 // If there is only a '/' a the beginning.
3630057 //
3630058 if (path[0] == '/' &&
3630059     strchr (&path[1], (unsigned int) '/') == NULL)
3630060 {
3630061     path[1] = 0;
3630062     return (path);
3630063 }
3630064 //
3630065 // Replace the last '/' with zero.
3630066 //
3630067 p = strrchr (path, (unsigned int) '/');
3630068 *p = 0;
3630069 //
3630070 // Now remove extra duplicated final '/', except the
3630071 // very first
3630072 // character: 'i' is kept greater than zero.
3630073 //
3630074 for (i = (strlen (path) - 1);
3630075     i > 0 && path[i] == '/'; i--)
3630076 {
3630077     path[i] = 0;
3630078 }
3630079 //
3630080 // Now 'path' appears as a reduced string: the
3630081 // original path string
3630082 // is modified.
3630083 //
3630084 return (path);
```

```
3630085 }
```

95.10 os32: «lib/netinet/icmp.h»

<<

Si veda la sezione [91.3](#).

```
3640001 #ifndef __NETINET_ICMP_H
3640002 #define __NETINET_ICMP_H      1
3640003 //-----
3640004 // GNU C compatible ICMPv4 header and definitions
3640005 //-----
3640006 #include <sys/types.h>
3640007 #include <netinet/in.h>
3640008 #include <netinet/ip.h>
3640009 //-----
3640010 struct icmphdr
3640011 {
3640012     uint8_t type; // message type [1]
3640013     uint8_t code; // type sub-code [2]
3640014     uint16_t checksum;
3640015     union
3640016     {
3640017         struct
3640018         {
3640019             uint16_t id;
3640020             uint16_t sequence;
3640021         } __attribute__((packed)) echo; // echo
3640022         // datagram
3640023         uint32_t gateway; // gateway address
3640024         struct
3640025         {
3640026             uint16_t unused;
3640027             uint16_t mtu;
3640028         } __attribute__((packed)) frag; // path mtu
3640029         // discovery
3640030     } un;
3640031 } __attribute__((packed));
```

```
3640032 //
3640033 // [1] message type:
3640034 //
3640035 #define ICMP_ECHOREPLY 0 // echo reply
3640036 #define ICMP_DEST_UNREACH 3 // destination
3640037 // unreachable
3640038 #define ICMP_SOURCE_QUENCH 4 // source
3640039 // quench
3640040 #define ICMP_REDIRECT 5 // redirect
3640041 // (change
3640042 // route)
3640043 #define ICMP_ECHO 8 // echo
3640044 // request
3640045 #define ICMP_TIME_EXCEEDED 11 // time
3640046 // exceeded
3640047 #define ICMP_PARAMETERPROB 12 // parameter
3640048 // problem
3640049 #define ICMP_TIMESTAMP 13 // timestamp
3640050 // request
3640051 #define ICMP_TIMESTAMPREPLY 14 // timestamp
3640052 // reply
3640053 #define ICMP_INFO_REQUEST 15 // information
3640054 // request
3640055 #define ICMP_INFO_REPLY 16 // information
3640056 // reply
3640057 #define ICMP_ADDRESS 17 // address
3640058 // mask
3640059 // request
3640060 #define ICMP_ADDRESSREPLY 18 // address
3640061 // mask reply
3640062 #define NR_ICMP_TYPES 18
3640063 //
3640064 // [2] type ICMP_DEST_UNREACH, code:
3640065 //
3640066 #define ICMP_NET_UNREACH 0 // network
3640067 // unreachable
3640068 #define ICMP_HOST_UNREACH 1 // host
```

```
3640069 // unreachable
3640070 #define ICMP_PROT_UNREACH 2 // protocol
3640071 // unreachable
3640072 #define ICMP_PORT_UNREACH 3 // port
3640073 // unreachable
3640074 #define ICMP_FRAG_NEEDED 4 // fragmentation
3640075 // needed/DF
3640076 // set
3640077 #define ICMP_SR_FAILED 5 // source
3640078 // route
3640079 // failed
3640080 #define ICMP_NET_UNKNOWN 6 // destination
3640081 // network
3640082 // unknown
3640083 #define ICMP_HOST_UNKNOWN 7 // destination
3640084 // host
3640085 // unknown
3640086 #define ICMP_HOST_ISOLATED 8 // source host
3640087 // isolated
3640088 #define ICMP_NET_ANO 9 // destination
3640089 // network
3640090 // administratively
3640091 // prohibited
3640092 #define ICMP_HOST_ANO 10 // destination
3640093 // host
3640094 // administratively
3640095 // prohibited
3640096 #define ICMP_NET_UNR_TOS 11 // network
3640097 // unreachable
3640098 // for this
3640099 // type of
3640100 // service
3640101 #define ICMP_HOST_UNR_TOS 12 // host
3640102 // unreachable
3640103 // for this
3640104 // type of
3640105 // service
```



```
3640106 #define ICMP_PKT_FILTERED 13 // packet
3640107 // filtered
3640108 #define ICMP_PREC_VIOLATION 14 // precedence
3640109 // violation
3640110 #define ICMP_PREC_CUTOFF 15 // precedence
3640111 // cut off
3640112 #define NR_ICMP_UNREACH 15 // instead of
3640113 // hardcoding
3640114 // immediate
3640115 // value
3640116 //
3640117 // [2] type ICMP_REDIRECT, code:
3640118 //
3640119 #define ICMP_REDIR_NET 0 // redirect
3640120 // net
3640121 #define ICMP_REDIR_HOST 1 // redirect
3640122 // host
3640123 #define ICMP_REDIR_NETTOS 2 // redirect
3640124 // net for TOS
3640125 #define ICMP_REDIR_HOSTTOS 3 // redirect
3640126 // host for
3640127 // TOS
3640128 //
3640129 // [2] type ICMP_TIME_EXCEEDED, code:
3640130 //
3640131 #define ICMP_EXC_TTL 0 // TTL count
3640132 // exceeded
3640133 #define ICMP_EXC_FRAGTIME 1 // fragment
3640134 // reass time
3640135 // exceeded
3640136 //-----
3640137 #endif
```

95.11 os32: «lib/netinet/in.h»

<<

Si veda la sezione [91.3](#).

```
3650001 #ifndef _NETINET_IN_H
3650002 #define _NETINET_IN_H      1
3650003 //-----
3650004 #include <stdint.h>
3650005 #include <sys/sa_family_t.h>
3650006 //-----
3650007 typedef uint16_t in_port_t;    // Port number. [1]
3650008 typedef uint32_t in_addr_t;   // IPv4 address.
3650009 //
3650010 // [1] Types 'in_port_t' and 'in_addr_t' are to be
3650011 //      intended for network byte order IPv4 integer
3650012 //      address, at least because this type is
3650013 //      used inside the type 'struct in_addr', that is
3650014 //      surely in network byte order. But attention must
3650015 //      be made to mistakes: for example,
3650016 //      inside the file <netinet/in.h> from GNU sources,
3650017 //      there are some macro defining default netmask
3650018 //      like this:
3650019 //
3650020 // #define IN_CLASSA(a)
3650021 //      (((in_addr_t) (a)) & 0x80000000) == 0)
3650022 // #define IN_CLASSB(a)
3650023 //      (((in_addr_t) (a)) & 0xc0000000) == 0x80000000)
3650024 // #define IN_CLASSC(a)
3650025 //      (((in_addr_t) (a)) & 0xe0000000) == 0xc0000000)
3650026 //
3650027 //      Such macro can work only if the architecture is
3650028 //      big-endian.
3650029 //
3650030 //-----
3650031 //
3650032 // IPv4 address.
3650033 //
3650034 struct in_addr
```

```
3650035 {
3650036     in_addr_t s_addr;
3650037 };
3650038 //
3650039 // struct sockaddr_in, members in *network*byte*order*.
3650040 //
3650041 struct sockaddr_in
3650042 {
3650043     sa_family_t sin_family;           // AF_INET.
3650044     in_port_t sin_port;               // Port number.
3650045     struct in_addr sin_addr;          // IP address.
3650046     uint8_t sin_zero[8];             // [2]
3650047 };
3650048 //
3650049 // [2] The type 'struct sockaddr_in' must be
3650050 //      replaceable with the type 'struct sockaddr',
3650051 //      with a cast. So it is necessary to fill the
3650052 //      unused space with a filler.
3650053 //
3650054 //-----
3650055 //
3650056 // IPv6 address, network byte order.
3650057 //
3650058 struct in6_addr
3650059 {
3650060     uint8_t s6_addr[16];
3650061 };
3650062 //
3650063 // struct sockaddr_in6, members in network byte order.
3650064 //
3650065 struct sockaddr_in6
3650066 {
3650067     sa_family_t sin6_family;          // AF_INET6.
3650068     in_port_t sin6_port;              // Port number.
3650069     uint32_t sin6_flowinfo;           // IPv6 traffic class
3650070     // and flow info.
3650071     struct in6_addr sin6_addr;        // IPv6 address.
```

```
3650072     uint32_t sin6_scope_id;           // Set of interfaces
3650073     // for a scope.
3650074 };
3650075 //-----
3650076 //external in6_addr in6addr_any;
3650077 //#define IN6ADDR_ANY_INIT ...
3650078 //external struct in6_addr in6addr_loopback;
3650079 //#define IN6ADDR_LOOPBACK_INIT ...
3650080 //-----
3650081 //
3650082 //
3650083 //
3650084 struct ipv6_mreq
3650085 {
3650086     struct in6_addr ipv6mr_multiaddr;    // IPv6
3650087     // multicast
3650088     // address.
3650089     unsigned int ipv6mr_interface;      // Interface
3650090     // index.
3650091 };
3650092 //-----
3650093 #define IPPROTO_IP          0           // Internet protocol.
3650094 #define IPPROTO_ICMP       1           // Contro message
3650095                               // protocol.
3650096 #define IPPROTO_TCP        6           // Transmission
3650097                               // control protocol.
3650098 #define IPPROTO_UDP        17          // User datagram
3650099                               // protocol.
3650100 #define IPPROTO_IPV6       41          // Internet protocol
3650101                               // version 6.
3650102 #define IPPROTO_RAW        255         // Raw IP packets
3650103                               // protocol
3650104 //-----
3650105 //
3650106 // 0.0.0.0
3650107 //
3650108 #define INADDR_ANY          ((in_addr_t) 0x00000000)
```

```

3650109 //
3650110 // 255.255.255.255
3650111 //
3650112 #define INADDR_BROADCAST ((in_addr_t) 0xffffffff)
3650113 //
3650114 // 127.0.0.1
3650115 //
3650116 #define INADDR_LOOPBACK ((in_addr_t) 0x7f000001)
3650117 //
3650118 //
3650119 //
3650120 #define INET_ADDRSTRLEN 16 // IPv4 address string
3650121 // size.
3650122 #define INET6_ADDRSTRLEN 46 // IPv6 address string
3650123 // size.
3650124 //-----
3650125 #endif

```

95.12 os32: «lib/netinet/ip.h»

Si veda la sezione [91.3](#).

```

3660001 #ifndef _NETINET_IP_H
3660002 #define _NETINET_IP_H 1
3660003 //-----
3660004 // GNU C compatible IPv4 header.
3660005 //-----
3660006 #include <netinet/in.h>
3660007 //-----
3660008 struct iphdr
3660009 {
3660010     uint16_t ihl:4, // header length / 4
3660011     version:4; // IP version
3660012     uint8_t tos; // type of service
3660013     uint16_t tot_len; // total packet length

```

```
3660014     uint16_t id;    // identification
3660015     uint16_t frag_off;    // fragment offset field
3660016     uint8_t ttl;    // time to live
3660017     uint8_t protocol;    // contained protocol
3660018     uint16_t check;    // header checksum
3660019     in_addr_t saddr;    // source IP address
3660020     in_addr_t daddr;    // destination IP address
3660021     //
3660022     // Options after this point.
3660023     //
3660024 };
3660025 //-----
3660026 #define IPVERSION      4        // IP version number
3660027 #define IP_MAXPACKET   65535    // maximum packet size
3660028 //
3660029 #define MAXTTL         255      // maximum time to
3660030                               // live (seconds)
3660031 #define IPDEFTTL      64        // default ttl, from
3660032                               // RFC 1340
3660033 #define IPFRAGTTL     60        // time to live for
3660034                               // fragments
3660035 #define IPTTLDEC       1        // subtracted when
3660036                               // forwarding
3660037 //
3660038 #define IP_MSS         576      // default maximum
3660039                               // segment size
3660040 //-----
3660041 #endif
```

95.13 os32: «lib/netinet/tcp.h»



Si veda la sezione [91.3](#).

```
3670001 #ifndef _NETINET_TCP_H
3670002 #define _NETINET_TCP_H 1
3670003 //-----
3670004 // GNU C compatible UDP header.
3670005 //-----
3670006 #include <sys/types.h>
3670007 //-----
3670008 struct tcphdr
3670009 {
3670010     uint16_t source;
3670011     uint16_t dest;
3670012     uint32_t seq;
3670013     uint32_t ack_seq;
3670014     uint16_t res1:4,
3670015             doff:4,
3670016             fin:1, syn:1, rst:1, psh:1, ack:1, urg:1, res2:2;
3670017     uint16_t window;
3670018     uint16_t check;
3670019     uint16_t urg_ptr;
3670020 };
3670021 //-----
3670022 // ATTENZIONE: per dare un significato allo stato di
3670023 // una connessione, occorre distinguere in che modo si
3670024 // trova inizialmente il socket:
3670025 // attivo o passivo (passivo quando rimane in ascolto
3670026 // per una connessione).
3670027 //
3670028 enum
3670029 {
3670030     TCP_LISTEN = 1,           // waiting a connection
3670031     // request
3670032     TCP_SYN_SENT, // SYN was sent, waiting from the
3670033     // response SYN
3670034     TCP_SYN_RECV, // SYN received, waiting for ACK
```

```
3670035 TCP_ESTABLISHED, // SYN sent, SYN received and
3670036 // ACK sent
3670037 TCP_FIN_WAIT1, // local close, FIN sent,
3670038 // waiting ACK or FIN
3670039 TCP_FIN_WAIT2, // FIN sent, ACK received,
3670040 // waiting FIN
3670041 TCP_CLOSE_WAIT, // FIN received, ACK sent,
3670042 // waiting local close
3670043 TCP_CLOSING, // FIN sent, FIN received, ACK sent,
3670044 // waiting ACK
3670045 TCP_LAST_ACK, // FIN received, ACK and FIN sent,
3670046 // waiting ACK
3670047 TCP_TIME_WAIT, // after TCP_LAST_ACK, wait a
3670048 // little and remove
3670049 TCP_CLOSE, // connection removed
3670050 TCP_RESET // connection reset (not standard)
3670051 };
3670052
3670053 #define TCPOPT_EOL 0
3670054 #define TCPOPT_NOP 1
3670055 #define TCPOPT_MAXSEG 2
3670056 #define TCPOLEN_MAXSEG 4
3670057 #define TCPOPT_WINDOW 3
3670058 #define TCPOLEN_WINDOW 3
3670059 #define TCPOPT_SACK_PERMITTED 4
3670060 #define TCPOLEN_SACK_PERMITTED 2
3670061 #define TCPOPT_SACK 5
3670062 #define TCPOPT_TIMESTAMP 8
3670063 #define TCPOLEN_TIMESTAMP 10
3670064 //-----
3670065 //
3670066 // TCP max segment size: IP_MSS - IP header size.
3670067 // Suppose to have a max IP header of 56 bytes,
3670068 // TCP_MSS == 520.
3670069 //
3670070 #define TCP_MSS 520
3670071 //-----
```



```
3670072 // LA STRUTTURA SEGUENTE È DA VALUTARE, forse conviene
3670073 // fare una tabella a parte per le connessioni TCP.
3670074 //
3670075 struct tcp_info
3670076 {
3670077     uint8_t tcpi_state;
3670078     uint8_t tcpi_ca_state;
3670079     uint8_t tcpi_retransmits;
3670080     uint8_t tcpi_probes;
3670081     uint8_t tcpi_backoff;
3670082     uint8_t tcpi_options;
3670083     uint8_t tcpi_snd_wscale:4, tcpi_rcv_wscale:4;
3670084
3670085     uint32_t tcpi_rto;
3670086     uint32_t tcpi_ato;
3670087     uint32_t tcpi_snd_mss;
3670088     uint32_t tcpi_rcv_mss;
3670089
3670090     uint32_t tcpi_unacked;
3670091     uint32_t tcpi_sacked;
3670092     uint32_t tcpi_lost;
3670093     uint32_t tcpi_retrans;
3670094     uint32_t tcpi_fackets;
3670095
3670096     /* Times. */
3670097     uint32_t tcpi_last_data_sent;
3670098
3670099     /* Not remembered, sorry. */
3670100     uint32_t tcpi_last_ack_sent;
3670101
3670102     uint32_t tcpi_last_data_recv;
3670103     uint32_t tcpi_last_ack_recv;
3670104
3670105     /* Metrics. */
3670106     uint32_t tcpi_pmtu;
3670107     uint32_t tcpi_rcv_ssthresh;
3670108     uint32_t tcpi_rtt;
```

```

3670109     uint32_t tcpi_rttvar;
3670110     uint32_t tcpi_snd_ssthresh;
3670111     uint32_t tcpi_snd_cwnd;
3670112     uint32_t tcpi_advmss;
3670113     uint32_t tcpi_reordering;
3670114
3670115     uint32_t tcpi_rcv_rtt;
3670116     uint32_t tcpi_rcv_space;
3670117
3670118     uint32_t tcpi_total_retrans;
3670119 };
3670120
3670121
3670122 //-----
3670123 #endif

```

95.14 os32: «lib/netinet/udp.h»

«

Si veda la sezione [91.3](#).

```

3680001 #ifndef __NETINET_UDP_H
3680002 #define __NETINET_UDP_H    1
3680003 //-----
3680004 // GNU C compatible UDP header.
3680005 //-----
3680006 #include <sys/types.h>
3680007 //-----
3680008 struct udphdr
3680009 {
3680010     uint16_t source;        // source port
3680011     uint16_t dest;         // destination port
3680012     uint16_t len;         // length
3680013     uint16_t check;       // checksum
3680014 } __attribute__((packed));
3680015 //-----

```

3680016	#endif
---------	--------

95.15 os32: «lib/pwd.h»

Si veda la sezione [91.3](#).



```

3690001 #ifndef _PWD_H
3690002 #define _PWD_H          1
3690003 //-----
3690004 #include <restrict.h>
3690005 #include <sys/types.h> // gid_t, uid_t
3690006 //-----
3690007 struct passwd
3690008 {
3690009     char *pw_name;
3690010     char *pw_passwd;
3690011     uid_t pw_uid;
3690012     gid_t pw_gid;
3690013     char *pw_gecos;
3690014     char *pw_dir;
3690015     char *pw_shell;
3690016 };
3690017 //-----
3690018 struct passwd *getpwent (void);
3690019 void setpwent (void);
3690020 void endpwent (void);
3690021 struct passwd *getpwnam (const char *name);
3690022 struct passwd *getpwuid (uid_t uid);
3690023 //-----
3690024
3690025 #endif

```

[95.15.1](#) lib/pwd/pwent.c 1884

95.15.1 lib/pwd/pwent.c

<<

Si veda la sezione [88.57](#).

```
3700001 #include <pwd.h>
3700002 #include <stdio.h>
3700003 #include <string.h>
3700004 #include <stdlib.h>
3700005 //-----
3700006 static char buffer[BUFSIZ];
3700007 static struct passwd pw;
3700008 static FILE *fp = NULL;
3700009 //-----
3700010 struct passwd *
3700011 getpwent (void)
3700012 {
3700013     void *pstatus;
3700014     char *char_uid;
3700015     char *char_gid;
3700016     //
3700017     if (fp == NULL)
3700018     {
3700019         fp = fopen ("/etc/passwd", "r");
3700020         if (fp == NULL)
3700021         {
3700022             return NULL;
3700023         }
3700024     }
3700025     //
3700026     pstatus = fgets (buffer, BUFSIZ, fp);
3700027     if (pstatus == NULL)
3700028     {
3700029         return (NULL);
3700030     }
3700031     //
3700032     // The parse is made with 'strtok()'. Please notice
3700033     // that
3700034     // 'strtok()' will not parse a line like the
```

```
3700035 // following:
3700036 // user::1001:233:...
3700037 // The password field *must* have something,
3700038 // otherwise the
3700039 // UID will take the password place.
3700040 // 'strtok()' will consider '::' the same as ':'!
3700041 //
3700042 pw.pw_name = strtok (buffer, ":");
3700043 pw.pw_passwd = strtok (NULL, ":");
3700044 char_uid = strtok (NULL, ":");
3700045 char_gid = strtok (NULL, ":");
3700046 pw.pw_gecos = strtok (NULL, ":");
3700047 pw.pw_dir = strtok (NULL, ":");
3700048 pw.pw_shell = strtok (NULL, "\n");
3700049 pw.pw_uid = (uid_t) atoi (char_uid);
3700050 pw.pw_gid = (gid_t) atoi (char_gid);
3700051 //
3700052 return (&pw);
3700053 }
3700054
3700055 //-----
3700056 void
3700057 endpwent (void)
3700058 {
3700059     int status;
3700060     //
3700061     if (fp != NULL)
3700062     {
3700063         status = fclose (fp);
3700064         if (status != 0)
3700065         {
3700066             perror (NULL);
3700067             fp = NULL;
3700068         }
3700069     }
3700070     else
3700071     {
3700072         ; // printf ("[%s] fclose (fp)\n",
```

```
3700072         // __func__);
3700073     }
3700074 }
3700075 }
3700076
3700077 //-----
3700078 void
3700079 setpwent (void)
3700080 {
3700081     if (fp != NULL)
3700082     {
3700083         rewind (fp);
3700084     }
3700085 }
3700086
3700087 //-----
3700088 struct passwd *
3700089 getpwnam (const char *name)
3700090 {
3700091     struct passwd *pw;
3700092     //
3700093     setpwent ();
3700094     //
3700095     for (;;)
3700096     {
3700097         pw = getpwent ();
3700098         if (pw == NULL)
3700099         {
3700100             return (NULL);
3700101         }
3700102         if (strcmp (pw->pw_name, name) == 0)
3700103         {
3700104             return (pw);
3700105         }
3700106     }
3700107 }
3700108
```

```
3700109 //-----
3700110 struct passwd *
3700111 getpwuid (uid_t uid)
3700112 {
3700113     struct passwd *pw;
3700114     //
3700115     setpwent ();
3700116     //
3700117     for (;;)
3700118     {
3700119         pw = getpwent ();
3700120         if (pw == NULL)
3700121             {
3700122                 return (NULL);
3700123             }
3700124         if (pw->pw_uid == uid)
3700125             {
3700126                 return (pw);
3700127             }
3700128     }
3700129 }
```

95.16 os32: «lib/setjmp.h»

Si veda la sezione [87.49](#).

```
3710001 #ifndef _SETJMP_H
3710002 #define _SETJMP_H      1
3710003 //-----
3710004 #include <sys/os32.h>
3710005 #include <NULL.h>
3710006 //-----
3710007 typedef struct
3710008 {
3710009     uint32_t  eax0;
3710010     uint32_t  ecx0;
3710011     uint32_t  edx0;
```

```
3710012     uint32_t ebx0;
3710013     uint32_t ebp0;
3710014     uint32_t esi0;
3710015     uint32_t edi0;
3710016     uint32_t ds0;
3710017     uint32_t es0;
3710018     uint32_t fs0;
3710019     uint32_t gs0;
3710020     uint32_t eip0;
3710021     uint32_t cs0;
3710022     uint32_t eflags0;
3710023     //
3710024     uint32_t eip1;
3710025     uint32_t syscallnr;
3710026     uint32_t msg_pointer;
3710027     uint32_t msg_size;
3710028     //
3710029     uint32_t env;
3710030     uint32_t ret;
3710031     uint32_t ebp1;
3710032     uint32_t eip2;
3710033     //
3710034 } jmp_stack_t;
3710035
3710036 typedef struct
3710037 {
3710038     uint32_t esp0;
3710039     uint32_t eax0;
3710040     uint32_t ecx0;
3710041     uint32_t edx0;
3710042     uint32_t ebx0;
3710043     uint32_t ebp0;
3710044     uint32_t esi0;
3710045     uint32_t edi0;
3710046     uint32_t ds0;
3710047     uint32_t es0;
3710048     uint32_t fs0;
```



```

3710049     uint32_t gs0;
3710050     uint32_t eip0;
3710051     uint32_t cs0;
3710052     uint32_t eflags0;
3710053     //
3710054     uint32_t eip1;
3710055     uint32_t syscallnr;
3710056     uint32_t msg_pointer;
3710057     uint32_t msg_size;
3710058     //
3710059     uint32_t env;
3710060     uint32_t ret;
3710061     uint32_t ebp1;
3710062     uint32_t eip2;
3710063     //
3710064 } jmp_env_t;
3710065 //
3710066 typedef char jmp_buf[sizeof (jmp_env_t)];
3710067 //-----
3710068 int setjmp (jmp_buf env);
3710069 void longjmp (jmp_buf env, int val);
3710070 //-----
3710071 #endif

```

[95.16.1 lib/setjmp/longjmp.c](#) 1889

[95.16.2 lib/setjmp/setjmp.s](#) 1890

95.16.1 lib/setjmp/longjmp.c

Si veda la sezione [87.49](#).

```

3720001 #include <sys/os32.h>
3720002 #include <setjmp.h>
3720003 //-----
3720004 void
3720005 longjmp (jmp_buf env, int val)

```

```

3720006 {
3720007     sysmsg_jump_t msg;
3720008     msg.env = env;
3720009     msg.ret = val;
3720010     sys (SYS_LONGJMP, &msg, sizeof msg);
3720011 }

```

95.16.2 lib/setjmp/setjmp.s

«

Si veda la sezione [87.49](#).

```

3730001 .global setjmp
3730002 .extern sys
3730003 #-----
3730004 .text
3730005 #-----
3730006 .align 4
3730007 setjmp:
3730008     #
3730009     # Previous pushes:
3730010     #
3730011     #   push &env
3730012     #   push back_address   # made by a call to
3730013     #                       # setjmp() function
3730014     #
3730015     enter $8, $0
3730016     #
3730017     # sysmsg_jump_t msg;
3730018     #
3730019     movl  $0,   -4(%ebp)      # msg.ret = 0;
3730020     #
3730021     movl  8(%ebp), %eax      # msg.env = env;
3730022     movl  %eax, -8(%ebp)
3730023     #
3730024     # sys (SYS_SETJMP, &msg, sizeof msg);
3730025     #
3730026     lea  -8(%ebp), %eax

```

```
3730027    pushl $8                # sizeof msg
3730028    pushl %eax             # &msg
3730029    pushl $47             # SYS_SETJMP
3730030    call  sys
3730031    add   $4, %esp
3730032    add   $4, %esp
3730033    add   $4, %esp
3730034    #
3730035    # return (msg.ret);
3730036    #
3730037    movl  -4(%ebp), %eax
3730038    leave
3730039    ret
```

95.17 os32: «lib/signal.h»

Si veda la sezione [91.3](#).



```
3740001    #ifndef _SIGNAL_H
3740002    #define _SIGNAL_H      1
3740003    //-----
3740004    #include <sys/types.h>
3740005    //-----
3740006    #define SIGHUP        1
3740007    #define SIGINT        2
3740008    #define SIGQUIT      3
3740009    #define SIGILL       4
3740010    #define SIGABRT      6
3740011    #define SIGFPE       8
3740012    #define SIGKILL      9
3740013    #define SIGSEGV     11
3740014    #define SIGPIPE     13
3740015    #define SIGALRM     14
3740016    #define SIGTERM     15
3740017    #define SIGSTOP     17
3740018    #define SIGTSTP     18
3740019    #define SIGCONT     19
```

```
3740020 #define SIGCHLD          20
3740021 #define SIGTTIN         21
3740022 #define SIGTTOU         22
3740023 #define SIGUSR1         30
3740024 #define SIGUSR2         31
3740025 //-----
3740026 typedef int sig_atomic_t;
3740027 typedef void (*sighandler_t) (int);      // [1]
3740028 //
3740029 // [1] The type 'sighandler_t' is a pointer to a
3740030 // function for the signal handling, with a parameter
3740031 // of type 'int', returning 'void'.
3740032 //
3740033 //-----
3740034 // Special function used to call the real signal
3740035 // handler. This function will return to the 'back'
3740036 // address, instead where it was called.
3740037 //
3740038 void _sighandler_wrapper (uint32_t handler,
3740039                          uint32_t signal, uint32_t back);
3740040 //-----
3740041 // Special undeclarable functions.
3740042 //
3740043 #define SIG_ERR ((sighandler_t) -1)      // [2]
3740044 #define SIG_DFL ((sighandler_t) 0)      // [2]
3740045 #define SIG_IGN ((sighandler_t) 1)      // [2]
3740046 //
3740047 // [2] It transforms an integer number into a
3740048 // 'sighandler_t' type, that is, a pointer
3740049 // to a function that does not exists really.
3740050 //
3740051 //-----
3740052 sighandler_t signal (int sig, sighandler_t handler);
3740053 int kill (pid_t pid, int sig);
3740054 int raise (int sig);
3740055 //-----
```

3740056	#endif
---------	--------

95.17.1	lib/signal/_sighandler_wrapper.s	1893
95.17.2	lib/signal/kill.c	1895
95.17.3	lib/signal/signal.c	1896

95.17.1 lib/signal/_sighandler_wrapper.s

Si veda la sezione [87.52](#).



```

3750001  .global _sighandler_wrapper
3750002  #-----
3750003  .section .text
3750004  #-----
3750005  # Port input byte.
3750006  #-----
3750007  _sighandler_wrapper:
3750008      #
3750009      # Current stack is:
3750010      #
3750011      # push %eip          # Back from interrupted code.
3750012      # push <sig_num>     # Signal number.
3750013      # push <sig_handler> # Signal handler address
3750014      #
3750015      # Please note that THERE IS NO RETURN ADDRESS!
3750016      # Instead you find the signal handler address
3750017      # there.
3750018      #
3750019      # This routine should have to call the signal
3750020      # handler function, and then return back to the
3750021      # interrupted code.
3750022      #
3750023      enter $0, $0          # No local variables.
3750024      pushf
3750025      pusha

```

```
3750026 .equ SIG_HAND, 4 # First argument. [1]
3750027 .equ SIG_NUM, 8 # Second argument. [1]
3750028 #
3750029 # [1] This function is called without the return
3750030 # address inside the stack. So the arguments
3750031 # are 4 bytes nearer than the usual.
3750032 #
3750033 mov SIG_NUM(%ebp), %edx # Copy the signal
3750034 # number into EDX.
3750035 mov SIG_HAND(%ebp), %eax # Copy the signal
3750036 # handler function
3750037 # address into EAX.
3750038 push %edx # Prepare argument for
3750039 # the signal
3750040 # handler function.
3750041 call *%eax # Call the signal
3750042 # handler function.
3750043 add $4, %esp # Pop the signal
3750044 # number argument.
3750045 popa
3750046 popf
3750047 leave
3750048 #
3750049 # Now we are back to the same stack as the
3750050 # beginning:
3750051 #
3750052 # push %eip # back from interrupted code.
3750053 # push <sig_num>
3750054 # push <sig_handler>
3750055 # push %eip # back from
3750056 # # _sighandler_wrapper()
3750057 #
3750058 # The stack pointer must be modified before
3750059 # returning, so that the address to the original
3750060 # interrupted instruction is used for return.
3750061 # Without such modification, the RET
3750062 # instruction would find the signal handler address
```

```
3750063     # instead!
3750064     #
3750065     add $4, %esp
3750066     add $4, %esp
3750067     #
3750068     # Now we are ready to return to the original
3750069     # interrupted address!
3750070     #
3750071     ret
3750072
```

95.17.2 lib/signal/kill.c

Si veda la sezione [87.29](#).



```
3760001 #include <sys/os32.h>
3760002 #include <sys/types.h>
3760003 #include <signal.h>
3760004 #include <errno.h>
3760005 #include <string.h>
3760006 //-----
3760007 int
3760008 kill (pid_t pid, int sig)
3760009 {
3760010     sysmsg_kill_t msg;
3760011     if (pid < -1) // Currently unsupported.
3760012     {
3760013         errset (ESRCH);
3760014         return (-1);
3760015     }
3760016     msg.pid = pid;
3760017     msg.signal = sig;
3760018     msg.ret = 0;
3760019     msg.errno = 0;
3760020     sys (SYS_KILL, &msg, (sizeof msg));
3760021     errno = msg.errno;
3760022     errln = msg.errln;
```

```
3760023     strncpy (errfn, msg.errfn, PATH_MAX);
3760024     return (msg.ret);
3760025 }
```

95.17.3 lib/signal/signal.c

«

Si veda la sezione [87.52](#).

```
3770001 #include <sys/os32.h>
3770002 #include <sys/types.h>
3770003 #include <signal.h>
3770004 #include <errno.h>
3770005 #include <string.h>
3770006 //-----
3770007 sighandler_t
3770008 signal (int sig, sighandler_t handler)
3770009 {
3770010     sysmsg_signal_t msg;
3770011
3770012     msg.signal = sig;
3770013     msg.handler = handler;
3770014     msg.wrapper = (uintptr_t) _sighandler_wrapper;
3770015     msg.ret = SIG_DFL;
3770016     msg.errno = 0;
3770017     sys (SYS_SIGNAL, &msg, (sizeof msg));
3770018     errno = msg.errno;
3770019     errln = msg.errln;
3770020     strncpy (errfn, msg.errfn, PATH_MAX);
3770021     return (msg.ret);
3770022 }
```


95.18 os32: «lib/stdio.h»



Si veda la sezione [88.112](#).

```
3780001 #ifndef _STDIO_H
3780002 #define _STDIO_H          1
3780003 //-----
3780004 #include <restrict.h>
3780005 #include <stdarg.h>
3780006 #include <stdint.h>
3780007 #include <limits.h>
3780008 #include <NULL.h>
3780009 #include <size_t.h>
3780010 #include <sys/types.h>
3780011 #include <SEEK.h>          // SEEK_CUR, SEEK_SET,
3780012                          // SEEK_END
3780013 //-----
3780014 #define BUFSIZ            8192 // At least the
3780015                          // file
3780016                          // system max zone
3780017                          // size.
3780018 #define _IOFBF            0 // Input-output
3780019                          // fully
3780020                          // buffered.
3780021 #define _IOLBF            1 // Input-output
3780022                          // line
3780023                          // buffered.
3780024 #define _IONBF            2 // Input-output
3780025                          // with
3780026                          // no buffering.
3780027
3780028 #define L_tmpnam          FILENAME_MAX // <limits.h>
3780029
3780030 #define FOPEN_MAX         OPEN_MAX // <limits.h>
3780031 #define FILENAME_MAX     NAME_MAX // <limits.h>
3780032 #define TMP_MAX           0x7FFF
3780033
3780034 #define EOF                (-1) // Must be a
```

```
3780035                                     // negative
3780036                                     // value.
3780037 //-----
3780038 typedef off_t fpos_t;    // 'off_t' defined in
3780039                          // <sys/types.h>.
3780040
3780041 typedef struct
3780042 {
3780043     int fdn;             // File descriptor number.
3780044     char error;         // Error indicator.
3780045     char eof;          // End of file indicator.
3780046 } FILE;
3780047
3780048 extern FILE _stream[];  // Defined inside
3780049                          // 'lib/stdio/FILE.c'.
3780050
3780051 #define stdin    (&_stream[0])
3780052 #define stdout  (&_stream[1])
3780053 #define stderr  (&_stream[2])
3780054 //-----
3780055 void clearerr (FILE * fp);
3780056 int fclose (FILE * fp);
3780057 int feof (FILE * fp);
3780058 int ferror (FILE * fp);
3780059 int fflush (FILE * fp);
3780060 int fgetc (FILE * fp);
3780061 int fgetpos (FILE * restrict fp, fpos_t * restrict pos);
3780062 char *fgets (char *restrict string, int n,
3780063             FILE * restrict fp);
3780064 int fileno (FILE * fp);
3780065 FILE *fopen (const char *path, const char *mode);
3780066 int fprintf (FILE * fp, char *restrict format, ...);
3780067 int fputc (int c, FILE * fp);
3780068 int fputs (const char *restrict string, FILE * restrict fp);
3780069 size_t fread (void *restrict buffer, size_t size,
3780070              size_t nmemb, FILE * restrict fp);
3780071 FILE *freopen (const char *restrict path,
```

```
3780072         const char *restrict mode,
3780073         FILE * restrict fp);
3780074 int fscanf (FILE * restrict fp,
3780075         const char *restrict format, ...);
3780076 int fseek (FILE * fp, long int offset, int whence);
3780077 int fsetpos (FILE * fp, fpos_t * pos);
3780078 long int ftell (FILE * fp);
3780079 off_t ftello (FILE * fp);
3780080 size_t fwrite (const void *restrict buffer,
3780081         size_t size, size_t nmemb,
3780082         FILE * restrict fp);
3780083 #define getc(p)      (fgetc (p))
3780084 int getchar (void);
3780085 char *gets (char *string);
3780086 void perror (const char *string);
3780087 int printf (const char *restrict format, ...);
3780088 #define putc(c, p) (fputc ((c), (p)))
3780089 int putchar (int c);
3780090 int puts (const char *string);
3780091 void rewind (FILE * fp);
3780092 int scanf (const char *restrict format, ...);
3780093 void setbuf (FILE * restrict fp, char *restrict buffer);
3780094 int setvbuf (FILE * restrict fp, char *restrict buffer,
3780095         int buf_mode, size_t size);
3780096 int snprintf (char *restrict string, size_t size,
3780097         const char *restrict format, ...);
3780098 int sprintf (char *restrict string,
3780099         const char *restrict format, ...);
3780100 int sscanf (char *restrict string,
3780101         const char *restrict format, ...);
3780102 int vfprintf (FILE * fp, char *restrict format,
3780103         va_list arg);
3780104 int vfscanf (FILE * restrict fp,
3780105         const char *restrict format, va_list arg);
3780106 int vprintf (const char *restrict format, va_list arg);
3780107 int vscanf (const char *restrict format, va_list ap);
3780108 int vsnprintf (char *restrict string, size_t size,
```

```

3780109         const char *restrict format, va_list arg);
3780110 int vsprintf (char *restrict string,
3780111             const char *restrict format, va_list arg);
3780112 int vsscanf (const char *string, const char *format,
3780113             va_list ap);
3780114 //-----
3780115 #endif

```

95.18.1	lib/stdio/FILE.c	1902
95.18.2	lib/stdio/clearerr.c	1903
95.18.3	lib/stdio/fclose.c	1903
95.18.4	lib/stdio/feof.c	1903
95.18.5	lib/stdio/ferror.c	1904
95.18.6	lib/stdio/fflush.c	1904
95.18.7	lib/stdio/fgetc.c	1905
95.18.8	lib/stdio/fgetpos.c	1906
95.18.9	lib/stdio/fgets.c	1906
95.18.10	lib/stdio/fileno.c	1908
95.18.11	lib/stdio/fopen.c	1908
95.18.12	lib/stdio/fprintf.c	1910
95.18.13	lib/stdio/fputc.c	1911
95.18.14	lib/stdio/fputs.c	1911
95.18.15	lib/stdio/fread.c	1912
95.18.16	lib/stdio/freopen.c	1913
95.18.17	lib/stdio/fscanf.c	1914

Sorgenti della libreria generale	1901
95.18.18 lib/stdio/fseek.c	1915
95.18.19 lib/stdio/fseeko.c	1915
95.18.20 lib/stdio/fsetpos.c	1916
95.18.21 lib/stdio/ftell.c	1917
95.18.22 lib/stdio/ftello.c	1917
95.18.23 lib/stdio/fwrite.c	1917
95.18.24 lib/stdio/getchar.c	1918
95.18.25 lib/stdio/gets.c	1919
95.18.26 lib/stdio/perror.c	1921
95.18.27 lib/stdio/printf.c	1922
95.18.28 lib/stdio/putchar.c	1922
95.18.29 lib/stdio/puts.c	1923
95.18.30 lib/stdio/rewind.c	1923
95.18.31 lib/stdio/scanf.c	1924
95.18.32 lib/stdio/setbuf.c	1924
95.18.33 lib/stdio/setvbuf.c	1924
95.18.34 lib/stdio/snprintf.c	1925
95.18.35 lib/stdio/sprintf.c	1925
95.18.36 lib/stdio/sscanf.c	1926
95.18.37 lib/stdio/vfprintf.c	1926
95.18.38 lib/stdio/vfscanf.c	1927
95.18.39 lib/stdio/vfsscanf.c	1928

95.18.40	lib/stdio/vprintf.c	1973
95.18.41	lib/stdio/vscanf.c	1974
95.18.42	lib/stdio/vsnprintf.c	1975
95.18.43	lib/stdio/vsprintf.c	2012
95.18.44	lib/stdio/vsscanf.c	2013

95.18.1 lib/stdio/FILE.c

<<

Si veda la sezione [91.3](#).

```
3790001 #include <stdio.h>
3790002 //
3790003 // There must be room for at least 'FOPEN_MAX'
3790004 // elements.
3790005 //
3790006 FILE _stream[FOPEN_MAX];
3790007 //-----
3790008 void
3790009 _stdio_stream_setup (void)
3790010 {
3790011     _stream[0].fdn = 0;
3790012     _stream[0].error = 0;
3790013     _stream[0].eof = 0;
3790014
3790015     _stream[1].fdn = 1;
3790016     _stream[1].error = 0;
3790017     _stream[1].eof = 0;
3790018
3790019     _stream[2].fdn = 2;
3790020     _stream[2].error = 0;
3790021     _stream[2].eof = 0;
3790022 }
```

95.18.2 lib/stdio/clearerr.c



Si veda la sezione [88.12](#).

```
3800001 #include <stdio.h>
3800002 //-----
3800003 void
3800004 clearerr (FILE * fp)
3800005 {
3800006     if (fp != NULL)
3800007     {
3800008         fp->error = 0;
3800009         fp->eof = 0;
3800010     }
3800011 }
```

95.18.3 lib/stdio/fclose.c



Si veda la sezione [88.28](#).

```
3810001 #include <stdio.h>
3810002 #include <unistd.h>
3810003 //-----
3810004 int
3810005 fclose (FILE * fp)
3810006 {
3810007     return (close (fp->fdn));
3810008 }
```

95.18.4 lib/stdio/feof.c



Si veda la sezione [88.29](#).

```
3820001 #include <stdio.h>
3820002 //-----
3820003 int
3820004 feof (FILE * fp)
3820005 {
```

```
3820006     if (fp != NULL)
3820007     {
3820008         return (fp->eof);
3820009     }
3820010     return (0);
3820011 }
```

95.18.5 lib/stdio/ferror.c

<<

Si veda la sezione [88.30](#).

```
3830001 #include <stdio.h>
3830002 //-----
3830003 int
3830004 ferror (FILE * fp)
3830005 {
3830006     if (fp != NULL)
3830007     {
3830008         return (fp->error);
3830009     }
3830010     return (0);
3830011 }
```

95.18.6 lib/stdio/fflush.c

<<

Si veda la sezione [88.31](#).

```
3840001 #include <stdio.h>
3840002 //-----
3840003 int
3840004 fflush (FILE * fp)
3840005 {
3840006     //
3840007     // The os32 library does not have any buffered data.
3840008     //
3840009     return (0);
3840010 }
```


95.18.7 lib/stdio/fgetc.c



Si veda la sezione [88.32](#).

```
3850001 #include <stdio.h>
3850002 #include <sys/types.h>
3850003 #include <unistd.h>
3850004 //-----
3850005 int
3850006 fgetc (FILE * fp)
3850007 {
3850008     ssize_t size_read;
3850009     int c;          // Character read.
3850010     //
3850011     for (c = 0;;)
3850012     {
3850013         size_read = read (fp->fdn, &c, (size_t) 1);
3850014         //
3850015         if (size_read <= 0)
3850016         {
3850017             //
3850018             // It is the end of file (zero) otherwise
3850019             // there is a
3850020             // problem (a negative value): return 'EOF'.
3850021             //
3850022             return (EOF);
3850023         }
3850024         //
3850025         // Valid read: end of scan.
3850026         //
3850027         return (c);
3850028     }
3850029 }
```

95.18.8 lib/stdio/fgetpos.c

<<

Si veda la sezione [88.33](#).

```
3860001 #include <stdio.h>
3860002 //-----
3860003 int
3860004 fgetpos (FILE * restrict fp, fpos_t * restrict pos)
3860005 {
3860006     long int position;
3860007     //
3860008     if (fp != NULL)
3860009     {
3860010         position = ftell (fp);
3860011         if (position >= 0)
3860012         {
3860013             *pos = position;
3860014             return (0);
3860015         }
3860016     }
3860017     return (-1);
3860018 }
```

95.18.9 lib/stdio/fgets.c

<<

Si veda la sezione [88.34](#).

```
3870001 #include <stdio.h>
3870002 #include <sys/types.h>
3870003 #include <unistd.h>
3870004 #include <stddef.h>
3870005 //-----
3870006 char *
3870007 fgets (char *restrict string, int n, FILE * restrict fp)
3870008 {
3870009     ssize_t size_read;
3870010     int b;           // Index inside the string buffer.
3870011     //
```

```
3870012     for (b = 0; b < (n - 1); b++, string[b] = 0)
3870013     {
3870014         size_read = read (fp->fdn, &string[b], (size_t) 1);
3870015         //
3870016         if (size_read <= 0)
3870017         {
3870018             //
3870019             // It is the end of file (zero) otherwise
3870020             // there is a
3870021             // problem (a negative value).
3870022             //
3870023             string[b] = 0;
3870024             break;
3870025         }
3870026         //
3870027         if (string[b] == '\n')
3870028         {
3870029             b++;
3870030             string[b] = 0;
3870031             break;
3870032         }
3870033     }
3870034     //
3870035     // If 'b' is zero, nothing was read and 'NULL' is
3870036     // returned.
3870037     //
3870038     if (b == 0)
3870039     {
3870040         return (NULL);
3870041     }
3870042     else
3870043     {
3870044         return (string);
3870045     }
3870046 }
```

95.18.10 lib/stdio/fileno.c



Si veda la sezione [88.35](#).

```
3880001 #include <stdio.h>
3880002 #include <errno.h>
3880003 //-----
3880004 int
3880005 fileno (FILE * fp)
3880006 {
3880007     if (fp != NULL)
3880008     {
3880009         return (fp->fdn);
3880010     }
3880011     errset (EBADF);           // Bad file descriptor.
3880012     return (-1);
3880013 }
```

95.18.11 lib/stdio/fopen.c



Si veda la sezione [88.36](#).

```
3890001 #include <fcntl.h>
3890002 #include <stdarg.h>
3890003 #include <stddef.h>
3890004 #include <string.h>
3890005 #include <errno.h>
3890006 #include <sys/os32.h>
3890007 #include <limits.h>
3890008 #include <stdio.h>
3890009 //-----
3890010 FILE *
3890011 fopen (const char *path, const char *mode)
3890012 {
3890013     int fdn;
3890014     //
3890015     if (strcmp (mode, "r") || strcmp (mode, "rb"))
3890016     {
```

```
3890017     fdn = open (path, O_RDONLY);
3890018     }
3890019     else if (strcmp (mode, "r+") ||
3890020             strcmp (mode, "r+b") || strcmp (mode, "rb+"))
3890021     {
3890022         fdn = open (path, O_RDWR);
3890023     }
3890024     else if (strcmp (mode, "w") || strcmp (mode, "wb"))
3890025     {
3890026         fdn = open (path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
3890027     }
3890028     else if (strcmp (mode, "w+") ||
3890029             strcmp (mode, "w+b") || strcmp (mode, "wb+"))
3890030     {
3890031         fdn = open (path, O_RDWR | O_CREAT | O_TRUNC, 0666);
3890032     }
3890033     else if (strcmp (mode, "a") || strcmp (mode, "ab"))
3890034     {
3890035         fdn =
3890036             open (path,
3890037                 O_WRONLY | O_APPEND | O_CREAT | O_TRUNC,
3890038                 0666);
3890039     }
3890040     else if (strcmp (mode, "a+") ||
3890041             strcmp (mode, "a+b") || strcmp (mode, "ab+"))
3890042     {
3890043         fdn =
3890044             open (path,
3890045                 O_RDWR | O_APPEND | O_CREAT | O_TRUNC, 0666);
3890046     }
3890047     else
3890048     {
3890049         errset (EINVAL); // Invalid argument.
3890050         return (NULL);
3890051     }
3890052     //
3890053     // Check the file descriptor returned.
```

```
3890054 //
3890055 if (fdn < 0)
3890056 {
3890057     //
3890058     // The variable 'errno' is already set.
3890059     //
3890060     errset (errno);
3890061     return (NULL);
3890062 }
3890063 //
3890064 // A valid file descriptor is available: convert it
3890065 // into a file
3890066 // stream. Please note that the file descriptor
3890067 // number must be
3890068 // saved inside the corresponding '_stream[]' array,
3890069 // because the
3890070 // file pointer do not have knowledge of the
3890071 // relative position
3890072 // inside the array.
3890073 //
3890074 _stream[fdn].fdn = fdn;           // Saved the file
3890075 // descriptor number.
3890076 //
3890077 return (&_stream[fdn]);         // Returned the file
3890078 // stream pointer.
3890079 }
```

95.18.12 lib/stdio/fprintf.c



Si veda la sezione [88.91](#).

```
3900001 #include <stdio.h>
3900002 //-----
3900003 int
3900004 fprintf (FILE * fp, char *restrict format, ...)
3900005 {
3900006     va_list ap;
```

```
3900007     va_start (ap, format);
3900008     return (vfprintf (fp, format, ap));
3900009 }
```

95.18.13 lib/stdio/fputc.c

Si veda la sezione [88.38](#).

```
3910001 #include <stdio.h>
3910002 #include <sys/types.h>
3910003 #include <sys/os32.h>
3910004 #include <string.h>
3910005 #include <unistd.h>
3910006 //-----
3910007 int
3910008 fputc (int c, FILE * fp)
3910009 {
3910010     ssize_t size_written;
3910011     char character = (char) c;
3910012     size_written = write (fp->fdn, &character, (size_t) 1);
3910013     if (size_written < 0)
3910014     {
3910015         fp->eof = 1;
3910016         return (EOF);
3910017     }
3910018     return (c);
3910019 }
```

95.18.14 lib/stdio/fputs.c

Si veda la sezione [88.39](#).

```
3920001 #include <stdio.h>
3920002 #include <string.h>
3920003 //-----
3920004 int
3920005 fputs (const char *restrict string, FILE * restrict fp)
```

```
3920006 {
3920007     int i;           // Index inside the string to be
3920008     // printed.
3920009     int status;
3920010
3920011     for (i = 0; i < strlen (string); i++)
3920012     {
3920013         status = fputc (string[i], fp);
3920014         if (status == EOF)
3920015         {
3920016             fp->eof = 1;
3920017             return (EOF);
3920018         }
3920019     }
3920020     return (0);
3920021 }
```

95.18.15 lib/stdio/fread.c



Si veda la sezione [88.40](#).

```
3930001 #include <unistd.h>
3930002 #include <stdio.h>
3930003 //-----
3930004 size_t
3930005 fread (void *restrict buffer, size_t size,
3930006        size_t nmemb, FILE * restrict fp)
3930007 {
3930008     ssize_t size_read;
3930009     size_read =
3930010         read (fp->fdn, buffer, (size_t) (size * nmemb));
3930011     if (size_read == 0)
3930012     {
3930013         fp->eof = 1;
3930014         return ((size_t) 0);
3930015     }
3930016     else if (size_read < 0)
```



```
3930017     {
3930018         fp->error = 1;
3930019         return ((size_t) 0);
3930020     }
3930021     else
3930022     {
3930023         return ((size_t) (size_read / size));
3930024     }
3930025 }
```

95.18.16 lib/stdio/freopen.c

Si veda la sezione [88.36](#).



```
3940001 #include <fcntl.h>
3940002 #include <stdarg.h>
3940003 #include <stddef.h>
3940004 #include <string.h>
3940005 #include <errno.h>
3940006 #include <sys/os32.h>
3940007 #include <limits.h>
3940008 #include <stdio.h>
3940009 //-----
3940010 FILE *
3940011 freopen (const char *restrict path,
3940012         const char *restrict mode, FILE * restrict fp)
3940013 {
3940014     int status;
3940015     FILE *fp_new;
3940016     //
3940017     if (fp == NULL)
3940018     {
3940019         return (NULL);
3940020     }
3940021     //
3940022     status = fclose (fp);
3940023     if (status != 0)
```

```
3940024     {
3940025         fp->error = 1;
3940026         return (NULL);
3940027     }
3940028     //
3940029     fp_new = fopen (path, mode);
3940030     //
3940031     if (fp_new == NULL)
3940032     {
3940033         return (NULL);
3940034     }
3940035     //
3940036     if (fp_new != fp)
3940037     {
3940038         fclose (fp_new);
3940039         return (NULL);
3940040     }
3940041     //
3940042     return (fp_new);
3940043 }
```

95.18.17 lib/stdio/fscanf.c



Si veda la sezione [88.102](#).

```
3950001 #include <stdio.h>
3950002 //-----
3950003 int
3950004 fscanf (FILE * restrict fp,
3950005         const char *restrict format, ...)
3950006 {
3950007     va_list ap;
3950008     va_start (ap, format);
3950009     return vfscanf (fp, format, ap);
3950010 }
```

95.18.18 lib/stdio/fseek.c



Si veda la sezione [88.44](#).

```
3960001 #include <stdio.h>
3960002 #include <unistd.h>
3960003 //-----
3960004 int
3960005 fseek (FILE * fp, long int offset, int whence)
3960006 {
3960007     off_t off_new;
3960008     off_new = lseek (fp->fdn, (off_t) offset, whence);
3960009     if (off_new < 0)
3960010     {
3960011         fp->error = 1;
3960012         return (-1);
3960013     }
3960014     else
3960015     {
3960016         fp->eof = 0;
3960017         return (0);
3960018     }
3960019 }
```

95.18.19 lib/stdio/fseeko.c



Si veda la sezione [88.44](#).

```
3970001 #include <stdio.h>
3970002 #include <unistd.h>
3970003 //-----
3970004 int
3970005 fseeko (FILE * fp, off_t offset, int whence)
3970006 {
3970007     off_t off_new;
3970008     off_new = lseek (fp->fdn, offset, whence);
3970009     if (off_new < 0)
3970010     {
```

```
3970011     fp->error = 1;
3970012     return (-1);
3970013 }
3970014 else
3970015 {
3970016     return (0);
3970017 }
3970018 }
```

95.18.20 lib/stdio/fsetpos.c

«

Si veda la sezione [88.33](#).

```
3980001 #include <stdio.h>
3980002 //-----
3980003 int
3980004 fsetpos (FILE * fp, fpos_t * pos)
3980005 {
3980006     long int position;
3980007     //
3980008     if (fp != NULL)
3980009     {
3980010         position = fseek (fp, (long int) *pos, SEEK_SET);
3980011         if (position >= 0)
3980012         {
3980013             *pos = position;
3980014             return (0);
3980015         }
3980016     }
3980017     return (-1);
3980018 }
```

95.18.21 lib/stdio/ftell.c



Si veda la sezione [88.47](#).

```
3990001 #include <stdio.h>
3990002 #include <unistd.h>
3990003 //-----
3990004 long int
3990005 ftell (FILE * fp)
3990006 {
3990007     return ((long int) lseek (fp->fdn, (off_t) 0, SEEK_CUR));
3990008 }
```

95.18.22 lib/stdio/ftello.c



Si veda la sezione [88.47](#).

```
4000001 #include <stdio.h>
4000002 #include <unistd.h>
4000003 //-----
4000004 off_t
4000005 ftello (FILE * fp)
4000006 {
4000007     return (lseek (fp->fdn, (off_t) 0, SEEK_CUR));
4000008 }
```

95.18.23 lib/stdio/fwrite.c



Si veda la sezione [88.49](#).

```
4010001 #include <unistd.h>
4010002 #include <stdio.h>
4010003 //-----
4010004 size_t
4010005 fwrite (const void *restrict buffer, size_t size,
4010006         size_t nmemb, FILE * restrict fp)
4010007 {
4010008     ssize_t size_written;
```

```
4010009     size_written =
4010010         write (fp->fdn, buffer, (size_t) (size * nmemb));
4010011     if (size_written < 0)
4010012     {
4010013         fp->error = 1;
4010014         return ((size_t) 0);
4010015     }
4010016     else
4010017     {
4010018         return ((size_t) (size_written / size));
4010019     }
4010020 }
```

95.18.24 lib/stdio/getchar.c



Si veda la sezione [88.32](#).

```
4020001 #include <stdio.h>
4020002 #include <sys/types.h>
4020003 #include <unistd.h>
4020004 //-----
4020005 int
4020006 getchar (void)
4020007 {
4020008     ssize_t size_read;
4020009     int c;          // Character read.
4020010     //
4020011     for (c = 0;;)
4020012     {
4020013         size_read = read (STDIN_FILENO, &c, (size_t) 1);
4020014         //
4020015         if (size_read <= 0)
4020016         {
4020017             //
4020018             // It is the end of file (zero) otherwise
4020019             // there is a
4020020             // problem (a negative value): return 'EOF'.
```

```

4020021         //
4020022         _stream[STDIN_FILENO].eof = 1;
4020023         return (EOF);
4020024     }
4020025     //
4020026     // Valid read.
4020027     //
4020028     if (size_read == 0)
4020029     {
4020030         //
4020031         // If no character is ready inside the
4020032         // keyboard buffer, just
4020033         // retry.
4020034         //
4020035         continue;
4020036     }
4020037     //
4020038     // End of scan.
4020039     //
4020040     return (c);
4020041 }
4020042 }

```

95.18.25 lib/stdio/gets.c

Si veda la sezione [88.34](#).

```

4030001 #include <stdio.h>
4030002 #include <sys/types.h>
4030003 #include <unistd.h>
4030004 #include <stddef.h>
4030005 //-----
4030006 char *
4030007 gets (char *string)
4030008 {
4030009     ssize_t size_read;
4030010     int b;         // Index inside the string buffer.

```

```
4030011 //
4030012 for (b = 0;; b++, string[b] = 0)
4030013 {
4030014     size_read =
4030015         read (STDIN_FILENO, &string[b], (size_t) 1);
4030016     //
4030017     if (size_read <= 0)
4030018     {
4030019         //
4030020         // It is the end of file (zero) otherwise
4030021         // there is a
4030022         // problem (a negative value).
4030023         //
4030024         _stream[STDIN_FILENO].eof = 1;
4030025         string[b] = 0;
4030026         break;
4030027     }
4030028     //
4030029     if (string[b] == '\n')
4030030     {
4030031         b++;
4030032         string[b] = 0;
4030033         break;
4030034     }
4030035 }
4030036 //
4030037 // If 'b' is zero, nothing was read and 'NULL' is
4030038 // returned.
4030039 //
4030040 if (b == 0)
4030041 {
4030042     return (NULL);
4030043 }
4030044 else
4030045 {
4030046     return (string);
4030047 }
```


4030048

}

95.18.26 lib/stdio/perror.c

Si veda la sezione [88.90](#).

```
4040001 #include <stdio.h>
4040002 #include <errno.h>
4040003 #include <stddef.h>
4040004 #include <string.h>
4040005 //-----
4040006 void
4040007 perror (const char *string)
4040008 {
4040009     //
4040010     // If errno is zero, there is nothing to show.
4040011     //
4040012     if (errno == 0)
4040013     {
4040014         return;
4040015     }
4040016     //
4040017     // Show the string if there is one.
4040018     //
4040019     if (string != NULL && strlen (string) > 0)
4040020     {
4040021         printf ("%s: ", string);
4040022     }
4040023     //
4040024     // Show the translated error.
4040025     //
4040026     if (errfn[0] != 0 && errln != 0)
4040027     {
4040028         printf ("[%s:%u:%i] %s\n",
4040029             errfn, errln, errno, strerror (errno));
4040030     }
4040031     else
```

```
4040032     {
4040033         printf ("%i] %s\n", errno, strerror (errno));
4040034     }
4040035 }
```

95.18.27 lib/stdio/printf.c

<<

Si veda la sezione [88.91](#).

```
4050001 #include <stdio.h>
4050002 //-----
4050003 int
4050004 printf (const char *restrict format, ...)
4050005 {
4050006     va_list ap;
4050007     va_start (ap, format);
4050008     return (vprintf (format, ap));
4050009 }
```

95.18.28 lib/stdio/putchar.c

<<

Si veda la sezione [88.38](#).

```
4060001 #include <stdio.h>
4060002 #include <sys/types.h>
4060003 #include <sys/os32.h>
4060004 #include <string.h>
4060005 #include <unistd.h>
4060006 //-----
4060007 int
4060008 putchar (int c)
4060009 {
4060010     return (fputc (c, stdout));
4060011 }
```

95.18.29 lib/stdio/puts.c



Si veda la sezione [88.39](#).

```
4070001 #include <stdio.h>
4070002 //-----
4070003 int
4070004 puts (const char *string)
4070005 {
4070006     int status;
4070007     status = printf ("%s\n", string);
4070008     if (status < 0)
4070009         {
4070010             return (EOF);
4070011         }
4070012     else
4070013         {
4070014             return (status);
4070015         }
4070016 }
```

95.18.30 lib/stdio/rewind.c



Si veda la sezione [88.100](#).

```
4080001 #include <stdio.h>
4080002 //-----
4080003 void
4080004 rewind (FILE * fp)
4080005 {
4080006     (void) fseek (fp, 0L, SEEK_SET);
4080007     fp->error = 0;
4080008 }
```

95.18.31 lib/stdio/scanf.c

<<

Si veda la sezione [88.102](#).

```
4090001 #include <stdio.h>
4090002 //-----
4090003 int
4090004 scanf (const char *restrict format, ...)
4090005 {
4090006     va_list ap;
4090007     va_start (ap, format);
4090008     return vfscanf (stdin, format, ap);
4090009 }
```

95.18.32 lib/stdio/setbuf.c

<<

Si veda la sezione [88.103](#).

```
4100001 #include <stdio.h>
4100002 //-----
4100003 void
4100004 setbuf (FILE * restrict fp, char *restrict buffer)
4100005 {
4100006     //
4100007     // The os32 library does not have any buffered data.
4100008     //
4100009     return;
4100010 }
```

95.18.33 lib/stdio/setvbuf.c

<<

Si veda la sezione [88.103](#).

```
4110001 #include <stdio.h>
4110002 //-----
4110003 int
4110004 setvbuf (FILE * restrict fp, char *restrict buffer,
4110005          int buf_mode, size_t size)
```

```
4110006 {
4110007     //
4110008     // The os32 library does not have any buffered data.
4110009     //
4110010     return (0);
4110011 }
```

95.18.34 lib/stdio/snprintf.c

Si veda la sezione [88.91](#).

```
4120001 #include <stdio.h>
4120002 #include <stdarg.h>
4120003 //-----
4120004 int
4120005 snprintf (char *restrict string, size_t size,
4120006           const char *restrict format, ...)
4120007 {
4120008     va_list ap;
4120009     va_start (ap, format);
4120010     return vsnprintf (string, size, format, ap);
4120011 }
```

95.18.35 lib/stdio/sprintf.c

Si veda la sezione [88.91](#).

```
4130001 #include <stdio.h>
4130002 #include <stdarg.h>
4130003 //-----
4130004 int
4130005 sprintf (char *restrict string,
4130006          const char *restrict format, ...)
4130007 {
4130008     va_list ap;
4130009     va_start (ap, format);
4130010     return vsnprintf (string, (size_t) BUFSIZ, format, ap);
```

```
4130011 }  
}
```

95.18.36 lib/stdio/sscanf.c

<<

Si veda la sezione [88.102](#).

```
4140001 #include <stdio.h>  
4140002 //-----  
4140003 int  
4140004 sscanf (char *restrict string,  
4140005         const char *restrict format, ...)  
4140006 {  
4140007     va_list ap;  
4140008     va_start (ap, format);  
4140009     return vsscanf (string, format, ap);  
4140010 }
```

95.18.37 lib/stdio/vfprintf.c

<<

Si veda la sezione [88.137](#).

```
4150001 #include <stdio.h>  
4150002 #include <sys/types.h>  
4150003 #include <sys/os32.h>  
4150004 #include <string.h>  
4150005 #include <unistd.h>  
4150006 //-----  
4150007 int  
4150008 vfprintf (FILE * fp, char *restrict format, va_list arg)  
4150009 {  
4150010     ssize_t size_written;  
4150011     size_t size;  
4150012     size_t size_total;  
4150013     int status;  
4150014     char string[BUFSIZ];  
4150015     char *buffer = string;  
4150016     //
```

```
4150017     buffer[0] = 0;
4150018     status = vsprintf (buffer, format, arg);
4150019     //
4150020     size = strlen (buffer);
4150021     if (size >= BUFSIZ)
4150022     {
4150023         size = BUFSIZ;
4150024     }
4150025     //
4150026     for (size_total = 0, size_written = 0;
4150027         size_total < size;
4150028         size_total += size_written, buffer += size_written)
4150029     {
4150030         size_written =
4150031             write (fp->fdn, buffer, size - size_total);
4150032         if (size_written < 0)
4150033         {
4150034             return (size_total);
4150035         }
4150036     }
4150037     return (size);
4150038 }
```

95.18.38 lib/stdio/vfscanf.c

Si veda la sezione [88.138](#).

```
4160001 #include <stdio.h>
4160002
4160003 //-----
4160004 int vfscanf (FILE * restrict fp, const char *string,
4160005             const char *restrict format, va_list ap);
4160006 //-----
4160007 int
4160008 vfscanf (FILE * restrict fp,
4160009         const char *restrict format, va_list ap)
4160010 {
```

```
4160011     return (vfsscanf (fp, NULL, format, ap));
4160012 }
4160013
4160014 //-----
```

95.18.39 lib/stdio/vfsscanf.c

<<

Si veda la sezione [88.138](#).

```
4170001 #include <stdint.h>
4170002 #include <stdbool.h>
4170003 #include <stdlib.h>
4170004 #include <string.h>
4170005 #include <stdio.h>
4170006 #include <stdarg.h>
4170007 #include <ctype.h>
4170008 #include <errno.h>
4170009 #include <stddef.h>
4170010 //-----
4170011 //
4170012 // This function is not standard and is able to do the
4170013 // work of both 'vfscanf()' and 'vsscanf()'.
4170014 //
4170015 //-----
4170016 #define WIDTH_MAX          64
4170017 //-----
4170018 static intmax_t strtointmax (const char *restrict
4170019                             string,
4170020                             const char **restrict
4170021                             endptr, int base,
4170022                             size_t max_width);
4170023 static int ass_or_eof (int consumed, int assigned);
4170024 //-----
4170025 int
4170026 vfsscanf (FILE * restrict fp, const char *string,
4170027           const char *restrict format, va_list ap)
4170028 {
```



```
4170029     int f = 0;      // Format index.
4170030     char buffer[BUFSIZ];
4170031     const char *input = string;    // Default.
4170032     const char *start = input;    // Default.
4170033     const char *restrict next = NULL;
4170034     int scanned = 0;
4170035     //
4170036     bool stream = 0;
4170037     bool flag_star = 0;
4170038     bool specifier = 0;
4170039     bool specifier_flags = 0;
4170040     bool specifier_width = 0;
4170041     bool specifier_type = 0;
4170042     bool inverted = 0;
4170043     //
4170044     char *ptr_char;
4170045     signed char *ptr_schar;
4170046     unsigned char *ptr_uchar;
4170047     short int *ptr_sshort;
4170048     unsigned short int *ptr_ushort;
4170049     int *ptr_sint;
4170050     unsigned int *ptr_uint;
4170051     long int *ptr_slong;
4170052     unsigned long int *ptr_ulong;
4170053     intmax_t *ptr_simax;
4170054     uintmax_t *ptr_uimax;
4170055     size_t *ptr_size;
4170056     ptrdiff_t *ptr_ptrdiff;
4170057     void **ptr_void;
4170058     //
4170059     size_t width;
4170060     char width_string[WIDTH_MAX + 1];
4170061     int w;      // Index inside width string.
4170062     int assigned = 0;    // Assignment counter.
4170063     int consumed = 0;    // Consumed counter.
4170064     //
4170065     intmax_t value_i;
```

```
4170066     uintmax_t value_u;
4170067     //
4170068     const char *end_format;
4170069     const char *end_input;
4170070     int count;      // Generic counter.
4170071     int index;     // Generic index.
4170072     bool ascii[128];
4170073     //
4170074     void *pstatus;
4170075     //
4170076     // Initialize some data.
4170077     //
4170078     width_string[0] = '\\0';
4170079     end_format = format + (strlen (format));
4170080     //
4170081     // Check arguments and find where input comes.
4170082     //
4170083     if (fp == NULL && (string == NULL || string[0] == 0))
4170084     {
4170085         errset (EINVAL); // Invalid argument.
4170086         return (EOF);
4170087     }
4170088     //
4170089     if (fp != NULL && string != NULL && string[0] != 0)
4170090     {
4170091         errset (EINVAL); // Invalid argument.
4170092         return (EOF);
4170093     }
4170094     //
4170095     if (fp != NULL)
4170096     {
4170097         stream = 1;
4170098     }
4170099     //
4170100     //
4170101     //
4170102     for (;;)
```

```
4170103     {
4170104         if (stream)
4170105             {
4170106                 pstatus = fgets (buffer, BUFSIZ, fp);
4170107                 //
4170108                 if (pstatus == NULL)
4170109                     {
4170110                         return (ass_or_eof (consumed, assigned));
4170111                     }
4170112                 //
4170113                 input = buffer;
4170114                 start = input;
4170115                 next = NULL;
4170116             }
4170117         //
4170118         // Calculate end input.
4170119         //
4170120         end_input = input + (strlen (input));
4170121         //
4170122         // Scan format and input strings. Index 'f' is
4170123         // not reset.
4170124         //
4170125         while (&format[f] < end_format && input < end_input)
4170126             {
4170127                 if (!specifier)
4170128                     {
4170129                         // -----
4170130                         // The context is not
4170131                         // inside a specifier.
4170132                         // -----
4170133                         if (isspace (format[f]))
4170134                             {
4170135                                 // ----- Space.
4170136                                 while (isspace (*input))
4170137                                     {
4170138                                         input++;
4170139                                     }
```

```
4170140 //
4170141 // Verify that the input string is
4170142 // not finished.
4170143 //
4170144 if (input[0] == 0)
4170145 {
4170146 //
4170147 // As the input string is
4170148 // finished, the format
4170149 // string index is not advanced,
4170150 // because there
4170151 // might be more spaces on the
4170152 // next line (if
4170153 // there is a next line, of
4170154 // course).
4170155 //
4170156 continue;
4170157 }
4170158 else
4170159 {
4170160 f++;
4170161 continue;
4170162 }
4170163 }
4170164 if (format[f] != '%')
4170165 {
4170166 // ----- Ordinary character.
4170167 if (format[f] == *input)
4170168 {
4170169 input++;
4170170 f++;
4170171 continue;
4170172 }
4170173 else
4170174 {
4170175 return (ass_or_eof
4170176 (consumed, assigned));
```

```
4170177         }
4170178     }
4170179     if (format[f] == '%' && format[f + 1] == '%')
4170180     {
4170181         // ----- Matching a literal '%'.
4170182         f++;
4170183         if (format[f] == *input)
4170184         {
4170185             input++;
4170186             f++;
4170187             continue;
4170188         }
4170189         else
4170190         {
4170191             return (ass_or_eof
4170192                     (consumed, assigned));
4170193         }
4170194     }
4170195     if (format[f] == '%')
4170196     {
4170197         // ----- Percent of a specifier.
4170198         f++;
4170199         specifier = 1;
4170200         specifier_flags = 1;
4170201         continue;
4170202     }
4170203 }
4170204 //
4170205 if (specifier && specifier_flags)
4170206 {
4170207     // -----
4170208     // The context is inside
4170209     // specifier flags.
4170210     // -----
4170211     if (format[f] == '*')
4170212     {
4170213         // ----- Assignment suppression star.
```

```
4170214         flag_star = 1;
4170215         f++;
4170216     }
4170217     else
4170218     {
4170219         // -----
4170220         // End of flags and begin of
4170221         // specifier length.
4170222         // -----
4170223         specifier_flags = 0;
4170224         specifier_width = 1;
4170225     }
4170226 }
4170227 //
4170228 if (specifier && specifier_width)
4170229 {
4170230     // -----
4170231     // The context is inside a
4170232     // specifier width.
4170233     // -----
4170234     for (w = 0;
4170235         format[f] >= '0'
4170236         && format[f] <= '9'
4170237         && w < WIDTH_MAX; w++)
4170238     {
4170239         width_string[w] = format[f];
4170240         f++;
4170241     }
4170242     width_string[w] = '\\0';
4170243     width = atoi (width_string);
4170244     if (width > WIDTH_MAX)
4170245     {
4170246         width = WIDTH_MAX;
4170247     }
4170248     //
4170249     // -----
4170250     // A zero width means an unspecified
```

```
4170251 // limit for the field
4170252 // length.
4170253 // -----
4170254 // End of spec. width and
4170255 // begin of spec. type.
4170256 // -----
4170257 specifier_width = 0;
4170258 specifier_type = 1;
4170259 }
4170260 //
4170261 if (specifier && specifier_type)
4170262 {
4170263 //
4170264 // Specifiers with length modifier.
4170265 //
4170266 if (format[f] == 'h' && format[f + 1] == 'h')
4170267 {
4170268 // ----- char.
4170269 if (format[f + 2] == 'd')
4170270 {
4170271 // ----- signed char, base 10.
4170272 value_i =
4170273     strtointmax (input, &next, 10,
4170274                 width);
4170275 if (input == next)
4170276 {
4170277     return (ass_or_eof
4170278           (consumed, assigned));
4170279 }
4170280 consumed++;
4170281 if (!flag_star)
4170282 {
4170283     ptr_schar =
4170284         va_arg (ap, signed char *);
4170285     *ptr_schar = value_i;
4170286     assigned++;
4170287 }
```

```
4170288         f += 3;
4170289         input = next;
4170290     }
4170291     else if (format[f + 2] == 'i')
4170292     {
4170293         // -----
4170294         // signed char, base unknown.
4170295         // -----
4170296         value_i =
4170297             strtointmax (input, &next, 0,
4170298                         width);
4170299         if (input == next)
4170300         {
4170301             return (ass_or_eof
4170302                     (consumed, assigned));
4170303         }
4170304         consumed++;
4170305         if (!flag_star)
4170306         {
4170307             ptr_schar =
4170308                 va_arg (ap, signed char *);
4170309             *ptr_schar = value_i;
4170310             assigned++;
4170311         }
4170312         f += 3;
4170313         input = next;
4170314     }
4170315     else if (format[f + 2] == 'o')
4170316     {
4170317         // -----
4170318         // signed char, base 8.
4170319         // -----
4170320         value_i =
4170321             strtointmax (input, &next, 8,
4170322                         width);
4170323         if (input == next)
4170324         {
```



```
4170325         return (ass_or_eof
4170326                 (consumed, assigned));
4170327     }
4170328     consumed++;
4170329     if (!flag_star)
4170330     {
4170331         ptr_schar =
4170332             va_arg (ap, signed char *);
4170333         *ptr_schar = value_i;
4170334         assigned++;
4170335     }
4170336     f += 3;
4170337     input = next;
4170338 }
4170339 else if (format[f + 2] == 'u')
4170340 {
4170341     // -----
4170342     // unsigned char, base 10.
4170343     // -----
4170344     value_u =
4170345         strtointmax (input, &next, 10,
4170346                     width);
4170347     if (input == next)
4170348     {
4170349         return (ass_or_eof
4170350                 (consumed, assigned));
4170351     }
4170352     consumed++;
4170353     if (!flag_star)
4170354     {
4170355         ptr_uchar =
4170356             va_arg (ap, unsigned char *);
4170357         *ptr_uchar = value_u;
4170358         assigned++;
4170359     }
4170360     f += 3;
4170361     input = next;
```

```
4170362     }
4170363     else if (format[f + 2] == 'x'
4170364             || format[f + 2] == 'X')
4170365     {
4170366         // -----
4170367         // signed char, base 16.
4170368         // -----
4170369         value_i =
4170370             strtointmax (input, &next, 16,
4170371                         width);
4170372         if (input == next)
4170373         {
4170374             return (ass_or_eof
4170375                     (consumed, assigned));
4170376         }
4170377         consumed++;
4170378         if (!flag_star)
4170379         {
4170380             ptr_schar =
4170381                 va_arg (ap, signed char *);
4170382             *ptr_schar = value_i;
4170383             assigned++;
4170384         }
4170385         f += 3;
4170386         input = next;
4170387     }
4170388     else if (format[f + 2] == 'n')
4170389     {
4170390         // -----
4170391         // signed char,
4170392         // string index counter.
4170393         // -----
4170394         ptr_schar =
4170395             va_arg (ap, signed char *);
4170396         *ptr_schar =
4170397             (signed char) (input - start +
4170398                           scanned);
```

```
4170399         f += 3;
4170400     }
4170401     else
4170402     {
4170403         // -----
4170404         // unsupported or
4170405         // unknown specifier.
4170406         // -----
4170407         f += 2;
4170408     }
4170409 }
4170410 else if (format[f] == 'h')
4170411 {
4170412     // ----- short.
4170413     if (format[f + 1] == 'd')
4170414     {
4170415         // -----
4170416         // signed short, base 10.
4170417         // -----
4170418         value_i =
4170419             strtointmax (input, &next, 10,
4170420                          width);
4170421         if (input == next)
4170422         {
4170423             return (ass_or_eof
4170424                    (consumed, assigned));
4170425         }
4170426         consumed++;
4170427         if (!flag_star)
4170428         {
4170429             ptr_sshort =
4170430                 va_arg (ap, signed short *);
4170431             *ptr_sshort = value_i;
4170432             assigned++;
4170433         }
4170434         f += 2;
4170435         input = next;
```

```
4170436     }
4170437     else if (format[f + 1] == 'i')
4170438     {
4170439         // -----
4170440         // signed
4170441         // short, base unknown.
4170442         // -----
4170443         value_i =
4170444             strtointmax (input, &next, 0,
4170445                         width);
4170446         if (input == next)
4170447         {
4170448             return (ass_or_eof
4170449                     (consumed, assigned));
4170450         }
4170451         consumed++;
4170452         if (!flag_star)
4170453         {
4170454             ptr_sshort =
4170455                 va_arg (ap, signed short *);
4170456             *ptr_sshort = value_i;
4170457             assigned++;
4170458         }
4170459         f += 2;
4170460         input = next;
4170461     }
4170462     else if (format[f + 1] == 'o')
4170463     {
4170464         // -----
4170465         // signed short, base 8.
4170466         // -----
4170467         value_i =
4170468             strtointmax (input, &next, 8,
4170469                         width);
4170470         if (input == next)
4170471         {
4170472             return (ass_or_eof
```

```
4170473                                     (consumed, assigned));
4170474     }
4170475     consumed++;
4170476     if (!flag_star)
4170477     {
4170478         ptr_sshort =
4170479             va_arg (ap, signed short *);
4170480         *ptr_sshort = value_i;
4170481         assigned++;
4170482     }
4170483     f += 2;
4170484     input = next;
4170485 }
4170486 else if (format[f + 1] == 'u')
4170487 {
4170488     // -----
4170489     // unsigned short, base 10.
4170490     // -----
4170491     value_u =
4170492         strtointmax (input, &next, 10,
4170493                     width);
4170494     if (input == next)
4170495     {
4170496         return (ass_or_eof
4170497                 (consumed, assigned));
4170498     }
4170499     consumed++;
4170500     if (!flag_star)
4170501     {
4170502         ptr_ushort =
4170503             va_arg (ap, unsigned short *);
4170504         *ptr_ushort = value_u;
4170505         assigned++;
4170506     }
4170507     f += 2;
4170508     input = next;
4170509 }
```



```
4170547     }
4170548     else
4170549     {
4170550         // -----
4170551         // unsupported or
4170552         // unknown specifier.
4170553         // -----
4170554         f += 1;
4170555     }
4170556 }
4170557 // ----- There is no 'long long int'.
4170558 else if (format[f] == 'l')
4170559 {
4170560     // ----- long int.
4170561     if (format[f + 1] == 'd')
4170562     {
4170563         // -----
4170564         // signed long, base 10.
4170565         // -----
4170566         value_i =
4170567             strtointmax (input, &next, 10,
4170568                         width);
4170569         if (input == next)
4170570         {
4170571             return (ass_or_eof
4170572                     (consumed, assigned));
4170573         }
4170574         consumed++;
4170575         if (!flag_star)
4170576         {
4170577             ptr_slong =
4170578                 va_arg (ap, signed long *);
4170579             *ptr_slong = value_i;
4170580             assigned++;
4170581         }
4170582         f += 2;
4170583         input = next;
```

```
4170584     }
4170585     else if (format[f + 1] == 'i')
4170586     {
4170587         // -----
4170588         // signed
4170589         // long, base unknown.
4170590         // -----
4170591         value_i =
4170592             strtointmax (input, &next, 0,
4170593                         width);
4170594         if (input == next)
4170595         {
4170596             return (ass_or_eof
4170597                     (consumed, assigned));
4170598         }
4170599         consumed++;
4170600         if (!flag_star)
4170601         {
4170602             ptr_slong =
4170603                 va_arg (ap, signed long *);
4170604             *ptr_slong = value_i;
4170605             assigned++;
4170606         }
4170607         f += 2;
4170608         input = next;
4170609     }
4170610     else if (format[f + 1] == 'o')
4170611     {
4170612         // -----
4170613         // signed long, base 8.
4170614         // -----
4170615         value_i =
4170616             strtointmax (input, &next, 8,
4170617                         width);
4170618         if (input == next)
4170619         {
4170620             return (ass_or_eof
```



```
4170621                                     (consumed, assigned));
4170622     }
4170623     consumed++;
4170624     if (!flag_star)
4170625     {
4170626         ptr_slong =
4170627             va_arg (ap, signed long *);
4170628         *ptr_slong = value_i;
4170629         assigned++;
4170630     }
4170631     f += 2;
4170632     input = next;
4170633 }
4170634 else if (format[f + 1] == 'u')
4170635 {
4170636     // -----
4170637     // unsigned long, base 10.
4170638     // -----
4170639     value_u =
4170640         strtointmax (input, &next, 10,
4170641                     width);
4170642     if (input == next)
4170643     {
4170644         return (ass_or_eof
4170645                 (consumed, assigned));
4170646     }
4170647     consumed++;
4170648     if (!flag_star)
4170649     {
4170650         ptr_ulong =
4170651             va_arg (ap, unsigned long *);
4170652         *ptr_ulong = value_u;
4170653         assigned++;
4170654     }
4170655     f += 2;
4170656     input = next;
4170657 }
```



```
4170695     }
4170696     else
4170697     {
4170698         // -----
4170699         // unsupported or
4170700         // unknown specifier.
4170701         // -----
4170702         f += 1;
4170703     }
4170704 }
4170705 else if (format[f] == 'j')
4170706 {
4170707     // ----- .----- intmax_t.
4170708     if (format[f + 1] == 'd')
4170709     {
4170710         // ----- intmax_t, base 10.
4170711         value_i =
4170712             strtointmax (input, &next, 10,
4170713                         width);
4170714         if (input == next)
4170715         {
4170716             return (ass_or_eof
4170717                     (consumed, assigned));
4170718         }
4170719         consumed++;
4170720         if (!flag_star)
4170721         {
4170722             ptr_simax =
4170723                 va_arg (ap, intmax_t *);
4170724             *ptr_simax = value_i;
4170725             assigned++;
4170726         }
4170727         f += 2;
4170728         input = next;
4170729     }
4170730     else if (format[f + 1] == 'i')
4170731     {
```

```
4170732 // -----
4170733 // intmax_t, base unknown.
4170734 // -----
4170735 value_i =
4170736     strtointmax (input, &next, 0,
4170737                 width);
4170738 if (input == next)
4170739     {
4170740         return (ass_or_eof
4170741                 (consumed, assigned));
4170742     }
4170743 consumed++;
4170744 if (!flag_star)
4170745     {
4170746         ptr_simax =
4170747             va_arg (ap, intmax_t *);
4170748         *ptr_simax = value_i;
4170749         assigned++;
4170750     }
4170751 f += 2;
4170752 input = next;
4170753 }
4170754 else if (format[f + 1] == 'o')
4170755     {
4170756         // -----
4170757         // intmax_t, base 8.
4170758         // -----
4170759 value_i =
4170760     strtointmax (input, &next, 8,
4170761                 width);
4170762 if (input == next)
4170763     {
4170764         return (ass_or_eof
4170765                 (consumed, assigned));
4170766     }
4170767 consumed++;
4170768 if (!flag_star)
```

```
4170769     {
4170770         ptr_simax =
4170771             va_arg (ap, intmax_t *);
4170772         *ptr_simax = value_i;
4170773         assigned++;
4170774     }
4170775     f += 2;
4170776     input = next;
4170777 }
4170778 else if (format[f + 1] == 'u')
4170779     {
4170780         // -----
4170781         // uintmax_t, base 10.
4170782         // -----
4170783         value_u =
4170784             strtointmax (input, &next, 10,
4170785                         width);
4170786         if (input == next)
4170787             {
4170788                 return (ass_or_eof
4170789                         (consumed, assigned));
4170790             }
4170791         consumed++;
4170792         if (!flag_star)
4170793             {
4170794                 ptr_uimax =
4170795                     va_arg (ap, uintmax_t *);
4170796                 *ptr_uimax = value_u;
4170797                 assigned++;
4170798             }
4170799         f += 2;
4170800         input = next;
4170801     }
4170802 else if (format[f + 1] == 'x'
4170803         || format[f + 2] == 'X')
4170804     {
4170805         // -----
```

```
4170806 // intmax_t, base 16.
4170807 // -----
4170808 value_i =
4170809     strtointmax (input, &next, 16,
4170810                 width);
4170811 if (input == next)
4170812     {
4170813         return (ass_or_eof
4170814                 (consumed, assigned));
4170815     }
4170816 consumed++;
4170817 if (!flag_star)
4170818     {
4170819         ptr_simax =
4170820             va_arg (ap, intmax_t *);
4170821         *ptr_simax = value_i;
4170822         assigned++;
4170823     }
4170824     f += 2;
4170825     input = next;
4170826 }
4170827 else if (format[f + 1] == 'n')
4170828     {
4170829         // -----
4170830         // signed char,
4170831         // string index counter.
4170832         // -----
4170833         ptr_simax = va_arg (ap, intmax_t *);
4170834         *ptr_simax =
4170835             (intmax_t) (input - start +
4170836                       scanned);
4170837         f += 2;
4170838     }
4170839 else
4170840     {
4170841         // -----
4170842         // unsupported or
```

```
4170843         // unknown specifier.
4170844         // -----
4170845         f += 1;
4170846     }
4170847 }
4170848 else if (format[f] == 'z')
4170849 {
4170850     // ----- size_t.
4170851     if (format[f + 1] == 'd')
4170852     {
4170853         // -----
4170854         // size_t, base 10.
4170855         // -----
4170856         value_i =
4170857             strtointmax (input, &next, 10,
4170858                         width);
4170859         if (input == next)
4170860         {
4170861             return (ass_or_eof
4170862                     (consumed, assigned));
4170863         }
4170864         consumed++;
4170865         if (!flag_star)
4170866         {
4170867             ptr_size = va_arg (ap, size_t *);
4170868             *ptr_size = value_i;
4170869             assigned++;
4170870         }
4170871         f += 2;
4170872         input = next;
4170873     }
4170874     else if (format[f + 1] == 'i')
4170875     {
4170876         // -----
4170877         // size_t, base unknown.
4170878         // -----
4170879         value_i =
```

```
4170880         strtointmax (input, &next, 0,
4170881                     width);
4170882     if (input == next)
4170883     {
4170884         return (ass_or_eof
4170885                 (consumed, assigned));
4170886     }
4170887     consumed++;
4170888     if (!flag_star)
4170889     {
4170890         ptr_size = va_arg (ap, size_t *);
4170891         *ptr_size = value_i;
4170892         assigned++;
4170893     }
4170894     f += 2;
4170895     input = next;
4170896 }
4170897 else if (format[f + 1] == 'o')
4170898 {
4170899     // -----
4170900     // size_t, base 8.
4170901     // -----
4170902     value_i =
4170903         strtointmax (input, &next, 8,
4170904                     width);
4170905     if (input == next)
4170906     {
4170907         return (ass_or_eof
4170908                 (consumed, assigned));
4170909     }
4170910     consumed++;
4170911     if (!flag_star)
4170912     {
4170913         ptr_size = va_arg (ap, size_t *);
4170914         *ptr_size = value_i;
4170915         assigned++;
4170916     }
```



```
4170917         f += 2;
4170918         input = next;
4170919     }
4170920     else if (format[f + 1] == 'u')
4170921     {
4170922         // -----
4170923         // size_t, base 10.
4170924         // -----
4170925         value_u =
4170926             strtointmax (input, &next, 10,
4170927                         width);
4170928         if (input == next)
4170929         {
4170930             return (ass_or_eof
4170931                     (consumed, assigned));
4170932         }
4170933         consumed++;
4170934         if (!flag_star)
4170935         {
4170936             ptr_size = va_arg (ap, size_t *);
4170937             *ptr_size = value_u;
4170938             assigned++;
4170939         }
4170940         f += 2;
4170941         input = next;
4170942     }
4170943     else if (format[f + 1] == 'x'
4170944             || format[f + 2] == 'X')
4170945     {
4170946         // -----
4170947         // size_t, base 16.
4170948         // -----
4170949         value_i =
4170950             strtointmax (input, &next, 16,
4170951                         width);
4170952         if (input == next)
4170953         {
```

```
4170954         return (ass_or_eof
4170955                 (consumed, assigned));
4170956     }
4170957     consumed++;
4170958     if (!flag_star)
4170959     {
4170960         ptr_size = va_arg (ap, size_t *);
4170961         *ptr_size = value_i;
4170962         assigned++;
4170963     }
4170964     f += 2;
4170965     input = next;
4170966 }
4170967 else if (format[f + 1] == 'n')
4170968 {
4170969     // -----
4170970     // signed char,
4170971     // string index counter.
4170972     // -----
4170973     ptr_size = va_arg (ap, size_t *);
4170974     *ptr_size =
4170975         (size_t) (input - start + scanned);
4170976     f += 2;
4170977 }
4170978 else
4170979 {
4170980     // -----
4170981     // unsupported or
4170982     // unknown specifier.
4170983     // -----
4170984     f += 1;
4170985 }
4170986 }
4170987 else if (format[f] == 't')
4170988 {
4170989     // ----- ptrdiff_t.
4170990     if (format[f + 1] == 'd')
```

```
4170991     {
4170992         // -----
4170993         // ptrdiff_t, base 10.
4170994         // -----
4170995         value_i =
4170996             strtointmax (input, &next, 10,
4170997                         width);
4170998         if (input == next)
4170999             {
4171000                 return (ass_or_eof
4171001                         (consumed, assigned));
4171002             }
4171003         consumed++;
4171004         if (!flag_star)
4171005             {
4171006                 ptr_ptrdiff =
4171007                     va_arg (ap, ptrdiff_t *);
4171008                 *ptr_ptrdiff = value_i;
4171009                 assigned++;
4171010             }
4171011         f += 2;
4171012         input = next;
4171013     }
4171014     else if (format[f + 1] == 'i')
4171015         {
4171016             // -----
4171017             // ptrdiff_t, base unknown.
4171018             // -----
4171019             value_i =
4171020                 strtointmax (input, &next, 0,
4171021                             width);
4171022             if (input == next)
4171023                 {
4171024                     return (ass_or_eof
4171025                             (consumed, assigned));
4171026                 }
4171027             consumed++;
```

```
4171028     if (!flag_star)
4171029     {
4171030         ptr_ptrdiff =
4171031             va_arg (ap, ptrdiff_t *);
4171032         *ptr_ptrdiff = value_i;
4171033         assigned++;
4171034     }
4171035     f += 2;
4171036     input = next;
4171037 }
4171038 else if (format[f + 1] == 'o')
4171039 {
4171040     // -----
4171041     // ptrdiff_t, base 8.
4171042     // -----
4171043     value_i =
4171044         strtointmax (input, &next, 8,
4171045                     width);
4171046     if (input == next)
4171047     {
4171048         return (ass_or_eof
4171049                 (consumed, assigned));
4171050     }
4171051     consumed++;
4171052     if (!flag_star)
4171053     {
4171054         ptr_ptrdiff =
4171055             va_arg (ap, ptrdiff_t *);
4171056         *ptr_ptrdiff = value_i;
4171057         assigned++;
4171058     }
4171059     f += 2;
4171060     input = next;
4171061 }
4171062 else if (format[f + 1] == 'u')
4171063 {
4171064     // -----
```

```
4171065 // ptrdiff_t, base 10.
4171066 // -----
4171067 value_u =
4171068     strtointmax (input, &next, 10,
4171069                 width);
4171070 if (input == next)
4171071     {
4171072         return (ass_or_eof
4171073                 (consumed, assigned));
4171074     }
4171075 consumed++;
4171076 if (!flag_star)
4171077     {
4171078         ptr_ptrdiff =
4171079             va_arg (ap, ptrdiff_t *);
4171080         *ptr_ptrdiff = value_u;
4171081         assigned++;
4171082     }
4171083     f += 2;
4171084     input = next;
4171085 }
4171086 else if (format[f + 1] == 'x'
4171087         || format[f + 2] == 'X')
4171088     {
4171089         // -----
4171090         // ptrdiff_t, base 16.
4171091         // -----
4171092         value_i =
4171093             strtointmax (input, &next, 16,
4171094                         width);
4171095         if (input == next)
4171096             {
4171097                 return (ass_or_eof
4171098                         (consumed, assigned));
4171099             }
4171100         consumed++;
4171101         if (!flag_star)
```

```
4171102         {
4171103             ptr_ptrdiff =
4171104                 va_arg (ap, ptrdiff_t *);
4171105             *ptr_ptrdiff = value_i;
4171106             assigned++;
4171107         }
4171108         f += 2;
4171109         input = next;
4171110     }
4171111     else if (format[f + 1] == 'n')
4171112     {
4171113         // -----
4171114         // signed char,
4171115         // string index counter.
4171116         // -----
4171117         ptr_ptrdiff =
4171118             va_arg (ap, ptrdiff_t *);
4171119         *ptr_ptrdiff =
4171120             (ptrdiff_t) (input - start +
4171121                         scanned);
4171122         f += 2;
4171123     }
4171124     else
4171125     {
4171126         // -----
4171127         // unsupported or
4171128         // unknown specifier.
4171129         // -----
4171130         f += 1;
4171131     }
4171132 }
4171133 //
4171134 // Specifiers with no length modifier.
4171135 //
4171136 if (format[f] == 'd')
4171137 {
4171138     // ----- signed short, base 10.
```

```
4171139     value_i =
4171140         strtointmax (input, &next, 10, width);
4171141     if (input == next)
4171142     {
4171143         return (ass_or_eof
4171144             (consumed, assigned));
4171145     }
4171146     consumed++;
4171147     if (!flag_star)
4171148     {
4171149         ptr_sshort =
4171150             va_arg (ap, signed short *);
4171151         *ptr_sshort = value_i;
4171152         assigned++;
4171153     }
4171154     f += 1;
4171155     input = next;
4171156 }
4171157 else if (format[f] == 'i')
4171158 {
4171159     // -----
4171160     // signed
4171161     // int, base unknown.
4171162     // -----
4171163     value_i =
4171164         strtointmax (input, &next, 0, width);
4171165     if (input == next)
4171166     {
4171167         return (ass_or_eof
4171168             (consumed, assigned));
4171169     }
4171170     consumed++;
4171171     if (!flag_star)
4171172     {
4171173         ptr_sint = va_arg (ap, signed int *);
4171174         *ptr_sint = value_i;
4171175         assigned++;
```

```
4171176         }
4171177         f += 1;
4171178         input = next;
4171179     }
4171180     else if (format[f] == 'o')
4171181     {
4171182         // -----
4171183         // signed int, base 8.
4171184         // -----
4171185         value_i =
4171186             strtointmax (input, &next, 8, width);
4171187         if (input == next)
4171188         {
4171189             return (ass_or_eof
4171190                     (consumed, assigned));
4171191         }
4171192         consumed++;
4171193         if (!flag_star)
4171194         {
4171195             ptr_sint = va_arg (ap, signed int *);
4171196             *ptr_sint = value_i;
4171197             assigned++;
4171198         }
4171199         f += 1;
4171200         input = next;
4171201     }
4171202     else if (format[f] == 'u')
4171203     {
4171204         // -----
4171205         // unsigned short, base 10.
4171206         // -----
4171207         value_u =
4171208             strtointmax (input, &next, 10, width);
4171209         if (input == next)
4171210         {
4171211             return (ass_or_eof
4171212                     (consumed, assigned));
```



```
4171213     }
4171214     consumed++;
4171215     if (!flag_star)
4171216     {
4171217         ptr_uint =
4171218             va_arg (ap, unsigned int *);
4171219         *ptr_uint = value_u;
4171220         assigned++;
4171221     }
4171222     f += 1;
4171223     input = next;
4171224 }
4171225 else if (format[f] == 'x' || format[f] == 'X')
4171226 {
4171227     // -----
4171228     // signed short, base 16.
4171229     // -----
4171230     value_i =
4171231         strtointmax (input, &next, 16, width);
4171232     if (input == next)
4171233     {
4171234         return (ass_or_eof
4171235             (consumed, assigned));
4171236     }
4171237     consumed++;
4171238     if (!flag_star)
4171239     {
4171240         ptr_sint = va_arg (ap, signed int *);
4171241         *ptr_sint = value_i;
4171242         assigned++;
4171243     }
4171244     f += 1;
4171245     input = next;
4171246 }
4171247 else if (format[f] == 'c')
4171248 {
4171249     // ----- char[].
```

```
4171250     if (width == 0)
4171251         width = 1;
4171252         //
4171253     if (!flag_star)
4171254         ptr_char = va_arg (ap, char *);
4171255         //
4171256     for (count = 0;
4171257         width > 0 && *input != 0;
4171258         width--, ptr_char++, input++)
4171259     {
4171260         if (!flag_star)
4171261             *ptr_char = *input;
4171262         //
4171263         count++;
4171264     }
4171265     //
4171266     if (count)
4171267         consumed++;
4171268     if (count && !flag_star)
4171269         assigned++;
4171270     //
4171271     f += 1;
4171272 }
4171273 else if (format[f] == 's')
4171274 {
4171275     // ----- string.
4171276     if (!flag_star)
4171277         ptr_char = va_arg (ap, char *);
4171278     //
4171279     for (count = 0;
4171280         !isspace (*input)
4171281         && *input != 0; ptr_char++, input++)
4171282     {
4171283         if (!flag_star)
4171284             *ptr_char = *input;
4171285         //
4171286         count++;
```

```
4171287     }
4171288     if (!flag_star)
4171289         *ptr_char = 0;
4171290     //
4171291     if (count)
4171292         consumed++;
4171293     if (count && !flag_star)
4171294         assigned++;
4171295     //
4171296     f += 1;
4171297 }
4171298 else if (format[f] == '[')
4171299 {
4171300     //
4171301     f++;
4171302     //
4171303     if (format[f] == '^')
4171304     {
4171305         inverted = 1;
4171306         f++;
4171307     }
4171308     else
4171309     {
4171310         inverted = 0;
4171311     }
4171312     //
4171313     // Reset ascii array.
4171314     //
4171315     for (index = 0; index < 128; index++)
4171316     {
4171317         ascii[index] = inverted;
4171318     }
4171319     //
4171320     //
4171321     //
4171322     for (count = 0;
4171323         &format[f] < end_format; count++)
```

```
4171324         {
4171325             if (format[f] == ']' && count > 0)
4171326                 {
4171327                     break;
4171328                 }
4171329             //
4171330             // Check for an interval.
4171331             //
4171332             if (format[f + 1] == '-'
4171333                 && format[f + 2] != ']'
4171334                 && format[f + 2] != 0)
4171335                 {
4171336                     //
4171337                     // Interval.
4171338                     //
4171339                     for (index = format[f];
4171340                         index <= format[f + 2];
4171341                         index++)
4171342                         {
4171343                             ascii[index] = !inverted;
4171344                         }
4171345                     f += 3;
4171346                     continue;
4171347                 }
4171348             //
4171349             // Single character.
4171350             //
4171351             index = format[f];
4171352             ascii[index] = !inverted;
4171353             f++;
4171354         }
4171355         //
4171356         // Is the scan correctly finished?.
4171357         //
4171358         if (format[f] != ']'')
4171359             {
4171360                 return (ass_or_eof
```

```
4171361                                     (consumed, assigned));
4171362     }
4171363     //
4171364     // The ascii table is populated.
4171365     //
4171366     if (width == 0)
4171367         width = SIZE_MAX;
4171368     //
4171369     // Scan the input string.
4171370     //
4171371     if (!flag_star)
4171372         ptr_char = va_arg (ap, char *);
4171373     //
4171374     for (count = 0;
4171375         width > 0 && *input != 0;
4171376         width--, ptr_char++, input++)
4171377     {
4171378         index = *input;
4171379         if (ascii[index])
4171380             {
4171381                 if (!flag_star)
4171382                     *ptr_char = *input;
4171383                 count++;
4171384             }
4171385         else
4171386             {
4171387                 break;
4171388             }
4171389     }
4171390     //
4171391     if (count)
4171392         consumed++;
4171393     if (count && !flag_star)
4171394         assigned++;
4171395     //
4171396     f += 1;
4171397 }
```

```
4171398     else if (format[f] == 'p')
4171399     {
4171400         // ----- void *.
4171401         value_i =
4171402             strtointmax (input, &next, 16, width);
4171403         if (input == next)
4171404             {
4171405                 return (ass_or_eof
4171406                     (consumed, assigned));
4171407             }
4171408         consumed++;
4171409         if (!flag_star)
4171410             {
4171411                 ptr_void = va_arg (ap, void **);
4171412                 *ptr_void = (void *) ((int) value_i);
4171413                 assigned++;
4171414             }
4171415         f += 1;
4171416         input = next;
4171417     }
4171418     else if (format[f] == 'n')
4171419     {
4171420         // -----
4171421         // signed char,
4171422         // string index counter.
4171423         // -----
4171424         ptr_sint = va_arg (ap, signed int *);
4171425         *ptr_sint =
4171426             (signed char) (input - start + scanned);
4171427         f += 1;
4171428     }
4171429     else
4171430     {
4171431         // -----
4171432         // unsupported or
4171433         // unknown specifier.
4171434         // -----
```

```
4171435         ;
4171436     }
4171437
4171438     // -----
4171439     // End of specifier.
4171440     // -----
4171441
4171442     width_string[0] = '\0';
4171443     specifier = 0;
4171444     specifier_flags = 0;
4171445     specifier_width = 0;
4171446     specifier_type = 0;
4171447     flag_star = 0;
4171448
4171449     }
4171450 }
4171451 //
4171452 // The format or the input string is terminated.
4171453 //
4171454 if (&format[f] < end_format && stream)
4171455 {
4171456     //
4171457     // Only the input string is finished, and
4171458     // the input comes
4171459     // from a stream, so another read will be
4171460     // done.
4171461     //
4171462     scanned += (int) (input - start);
4171463     continue;
4171464 }
4171465 //
4171466 // The format string is terminated.
4171467 //
4171468 return (ass_or_eof (consumed, assigned));
4171469 }
4171470 }
4171471
```

```
4171472 //-----
4171473 static intmax_t
4171474 strtointmax (const char *restrict string,
4171475             const char **restrict endptr, int base,
4171476             size_t max_width)
4171477 {
4171478     int i;
4171479     int d;          // Digits counter.
4171480     int sign = +1;
4171481     intmax_t number;
4171482     intmax_t previous;
4171483     int digit;
4171484     //
4171485     bool flag_prefix_oct = 0;
4171486     bool flag_prefix_exa = 0;
4171487     bool flag_prefix_dec = 0;
4171488     //
4171489     // If the 'max_width' value is zero, fix it to the
4171490     // maximum
4171491     // that it can represent.
4171492     //
4171493     if (max_width == 0)
4171494     {
4171495         max_width = SIZE_MAX;
4171496     }
4171497     //
4171498     // Eat initial spaces, but if there are spaces,
4171499     // there is an
4171500     // error inside the calling function!
4171501     //
4171502     for (i = 0; isspace (string[i]); i++)
4171503     {
4171504         fprintf (stderr,
4171505                 "libc error: file \"%s\", line %i\n",
4171506                 __FILE__, __LINE__);
4171507     }
4171508 }
```



```
4171509 //
4171510 // Check sign. The 'max_width' counts also the sign,
4171511 // if there is
4171512 // one.
4171513 //
4171514 if (string[i] == '+')
4171515 {
4171516     sign = +1;
4171517     i++;
4171518     max_width--;
4171519 }
4171520 else if (string[i] == '-')
4171521 {
4171522     sign = -1;
4171523     i++;
4171524     max_width--;
4171525 }
4171526 //
4171527 // Check for prefix.
4171528 //
4171529 if (string[i] == '0')
4171530 {
4171531     if (string[i + 1] == 'x' || string[i + 1] == 'X')
4171532     {
4171533         flag_prefix_exa = 1;
4171534     }
4171535     if (isdigit (string[i + 1]))
4171536     {
4171537         flag_prefix_oct = 1;
4171538     }
4171539 }
4171540 //
4171541 if (string[i] > '0' && string[i] <= '9')
4171542 {
4171543     flag_prefix_dec = 1;
4171544 }
4171545 //
```

```
4171546 // Check compatibility with requested base.
4171547 //
4171548 if (flag_prefix_exa)
4171549     {
4171550         if (base == 0)
4171551             {
4171552                 base = 16;
4171553             }
4171554         else if (base == 16)
4171555             {
4171556                 ; // Ok.
4171557             }
4171558         else
4171559             {
4171560                 //
4171561                 // Incompatible sequence: only the initial
4171562                 // zero is reported.
4171563                 //
4171564                 *endptr = &string[i + 1];
4171565                 return ((intmax_t) 0);
4171566             }
4171567             //
4171568             // Move on, after the '0x' prefix.
4171569             //
4171570             i += 2;
4171571         }
4171572 //
4171573 if (flag_prefix_oct)
4171574     {
4171575         if (base == 0)
4171576             {
4171577                 base = 8;
4171578             }
4171579             //
4171580             // Move on, after the '0' prefix.
4171581             //
4171582             i += 1;
```

```
4171583     }
4171584     //
4171585     if (flag_prefix_dec)
4171586     {
4171587         if (base == 0)
4171588         {
4171589             base = 10;
4171590         }
4171591     }
4171592     //
4171593     // Scan the string.
4171594     //
4171595     for (d = 0, number = 0;
4171596         d < max_width && string[i] != 0; i++, d++)
4171597     {
4171598         if (string[i] >= '0' && string[i] <= '9')
4171599         {
4171600             digit = string[i] - '0';
4171601         }
4171602         else if (string[i] >= 'A' && string[i] <= 'F')
4171603         {
4171604             digit = string[i] - 'A' + 10;
4171605         }
4171606         else if (string[i] >= 'a' && string[i] <= 'f')
4171607         {
4171608             digit = string[i] - 'a' + 10;
4171609         }
4171610         else
4171611         {
4171612             digit = 999;
4171613         }
4171614         //
4171615         // Give a sign to the digit.
4171616         //
4171617         digit *= sign;
4171618         //
4171619         // Compare with the base.
```

```
4171620 //
4171621 if (base > (digit * sign))
4171622 {
4171623 //
4171624 // Check if the current digit can be safely
4171625 // computed.
4171626 //
4171627 previous = number;
4171628 number *= base;
4171629 number += digit;
4171630 if (number / base != previous)
4171631 {
4171632 //
4171633 // Out of range.
4171634 //
4171635 *endptr = &string[i + 1];
4171636 errset (ERANGE); // Result too large.
4171637 if (sign > 0)
4171638 {
4171639 return (INTMAX_MAX);
4171640 }
4171641 else
4171642 {
4171643 return (INTMAX_MIN);
4171644 }
4171645 }
4171646 }
4171647 else
4171648 {
4171649 *endptr = &string[i];
4171650 return (number);
4171651 }
4171652 }
4171653 //
4171654 // The string is finished or the max digits length
4171655 // is reached.
4171656 //
```

```
4171657     *endptr = &string[i];
4171658     //
4171659     return (number);
4171660 }
4171661
4171662 //-----
4171663 static int
4171664 ass_or_eof (int consumed, int assigned)
4171665 {
4171666     if (consumed == 0)
4171667     {
4171668         return (EOF);
4171669     }
4171670     else
4171671     {
4171672         return (assigned);
4171673     }
4171674 }
4171675
4171676 //-----
```

95.18.40 lib/stdio/vprintf.c

Si veda la sezione [88.137](#).

```
4180001 #include <stdio.h>
4180002 #include <sys/types.h>
4180003 #include <sys/os32.h>
4180004 #include <string.h>
4180005 #include <unistd.h>
4180006 //-----
4180007 int
4180008 vprintf (const char *restrict format, va_list arg)
4180009 {
4180010     ssize_t size_written;
4180011     size_t size;
4180012     size_t size_total;
```



```
4180013 int status;
4180014 char string[BUFSIZ];
4180015 char *buffer = string;
4180016
4180017 buffer[0] = 0;
4180018 status = vsprintf (buffer, format, arg);
4180019
4180020 size = strlen (buffer);
4180021 if (size >= BUFSIZ)
4180022     {
4180023         size = BUFSIZ;
4180024     }
4180025
4180026 for (size_total = 0, size_written = 0;
4180027     size_total < size;
4180028     size_total += size_written, buffer += size_written)
4180029     {
4180030         //
4180031         // Write to the standard output: file descriptor
4180032         // n. 1.
4180033         //
4180034         size_written =
4180035             write (STDOUT_FILENO, buffer, size - size_total);
4180036         if (size_written < 0)
4180037             {
4180038                 return (size_total);
4180039             }
4180040     }
4180041 return (size);
4180042 }
```

95.18.41 lib/stdio/vscanf.c



Si veda la sezione [88.138](#).

```
4190001 #include <stdio.h>
4190002 //-----
```

```
4190003 int
4190004 vscanf (const char *restrict format, va_list ap)
4190005 {
4190006     return (vfscanf (stdin, format, ap));
4190007 }
4190008
4190009 //-----
```

95.18.42 lib/stdio/vsnprintf.c

Si veda la sezione [88.137](#).

```
4200001 #include <stdint.h>
4200002 #include <stdbool.h>
4200003 #include <stdlib.h>
4200004 #include <string.h>
4200005 #include <stdio.h>
4200006 //-----
4200007 static size_t uimaxtoa (uintmax_t integer,
4200008                        char *buffer, int base,
4200009                        int uppercase, size_t size);
4200010 static size_t imaxtoa (intmax_t integer, char *buffer,
4200011                      int base, int uppercase,
4200012                      size_t size);
4200013 static size_t simaxtoa (intmax_t integer, char *buffer,
4200014                       int base, int uppercase,
4200015                       size_t size);
4200016 static size_t uimaxtoa_fill (uintmax_t integer,
4200017                             char *buffer, int base,
4200018                             int uppercase, int width,
4200019                             int filler, int max);
4200020 static size_t imaxtoa_fill (intmax_t integer,
4200021                             char *buffer, int base,
4200022                             int uppercase, int width,
4200023                             int filler, int max);
4200024 static size_t simaxtoa_fill (intmax_t integer,
4200025                             char *buffer, int base,
```

```
4200026         int uppercase, int width,
4200027         int filler, int max);
4200028 static size_t strtostr_fill (char *string,
4200029         char *buffer, int width,
4200030         int filler, int max);
4200031 //-----
4200032 int
4200033 vsnprintf (char *restrict string, size_t size,
4200034         const char *restrict format, va_list ap)
4200035 {
4200036     //
4200037     // We produce at most 'size-1' characters, + '\0'.
4200038     // 'size' is used also as the max size for internal
4200039     // strings, but only if it is not too big.
4200040     //
4200041     int f = 0;
4200042     int s = 0;
4200043     int remain = size - 1;
4200044     //
4200045     bool specifier = 0;
4200046     bool specifier_flags = 0;
4200047     bool specifier_width = 0;
4200048     bool specifier_precision = 0;
4200049     bool specifier_type = 0;
4200050     //
4200051     bool flag_plus = 0;
4200052     bool flag_minus = 0;
4200053     bool flag_space = 0;
4200054     bool flag_alternate = 0;
4200055     bool flag_zero = 0;
4200056     //
4200057     int alignment;
4200058     int filler;
4200059     //
4200060     intmax_t value_i;
4200061     uintmax_t value_ui;
4200062     char *value_cp;
```



```
420063 //
420064 size_t width;
420065 size_t precision;
420066 size_t str_size =
420067     (size > (BUFSIZ / 2) ? (BUFSIZ / 2) : size);
420068 char width_string[str_size];
420069 char precision_string[str_size];
420070 int w;
420071 int p;
420072 //
420073 width_string[0] = '\0';
420074 precision_string[0] = '\0';
420075 //
420076 while (format[f] != 0 && s < (size - 1))
420077     {
420078     if (!specifier)
420079         {
420080         // ----- The context is not
420081         // inside a specifier.
420082         if (format[f] != '%')
420083             {
420084             string[s] = format[f];
420085             s++;
420086             remain--;
420087             f++;
420088             continue;
420089             }
420090         if (format[f] == '%' && format[f + 1] == '%')
420091             {
420092             string[s] = '%';
420093             f++;
420094             f++;
420095             s++;
420096             remain--;
420097             continue;
420098             }
420099         if (format[f] == '%')
```

```
4200100         {
4200101             f++;
4200102             specifier = 1;
4200103             specifier_flags = 1;
4200104             continue;
4200105         }
4200106     }
4200107     //
4200108     if (specifier && specifier_flags)
4200109     {
4200110         // ----- The context is inside
4200111         // specifier flags.
4200112         if (format[f] == '+')
4200113         {
4200114             flag_plus = 1;
4200115             f++;
4200116             continue;
4200117         }
4200118         else if (format[f] == '-')
4200119         {
4200120             flag_minus = 1;
4200121             f++;
4200122             continue;
4200123         }
4200124         else if (format[f] == ' ')
4200125         {
4200126             flag_space = 1;
4200127             f++;
4200128             continue;
4200129         }
4200130         else if (format[f] == '#')
4200131         {
4200132             flag_alternate = 1;
4200133             f++;
4200134             continue;
4200135         }
4200136         else if (format[f] == '0')
```

```
4200137     {
4200138         flag_zero = 1;
4200139         f++;
4200140         continue;
4200141     }
4200142     else
4200143     {
4200144         specifier_flags = 0;
4200145         specifier_width = 1;
4200146     }
4200147 }
4200148 //
4200149 if (specifier && specifier_width)
4200150 {
4200151     // ----- The context is inside
4200152     // specifier width.
4200153     for (w = 0;
4200154          format[f] >= '0' && format[f] <= '9'
4200155          && w < str_size; w++)
4200156     {
4200157         width_string[w] = format[f];
4200158         f++;
4200159     }
4200160     width_string[w] = '\\0';
4200161
4200162     specifier_width = 0;
4200163
4200164     if (format[f] == '.')
4200165     {
4200166         specifier_precision = 1;
4200167         f++;
4200168     }
4200169     else
4200170     {
4200171         specifier_precision = 0;
4200172         specifier_type = 1;
4200173     }
```

```
4200174     }
4200175     //
4200176     if (specifier && specifier_precision)
4200177     {
4200178         // ----- The context is inside
4200179         // specifier precision.
4200180         for (p = 0;
4200181             format[f] >= '0' && format[f] <= '9'
4200182             && p < str_size; p++)
4200183         {
4200184             precision_string[p] = format[f];
4200185             p++;
4200186         }
4200187         precision_string[p] = '\\0';
4200188
4200189         specifier_precision = 0;
4200190         specifier_type = 1;
4200191     }
4200192     //
4200193     if (specifier && specifier_type)
4200194     {
4200195         // ----- The context is
4200196         // inside specifier type.
4200197         width = atoi (width_string);
4200198         precision = atoi (precision_string);
4200199         filler = ' ';
4200200         if (flag_zero)
4200201             filler = '0';
4200202         if (flag_space)
4200203             filler = ' ';
4200204         alignment = width;
4200205         if (flag_minus)
4200206         {
4200207             alignment = -alignment;
4200208             filler = ' ';    // The filler
4200209             // character cannot
4200210             // be zero, so it is black.
```

```
4200211     }
4200212     //
4200213     if (format[f] == 'h' && format[f + 1] == 'h')
4200214     {
4200215         if (format[f + 2] == 'd'
4200216             || format[f + 2] == 'i')
4200217         {
4200218             // -----
4200219             // signed char, base 10.
4200220             value_i = va_arg (ap, int);
4200221             if (flag_plus)
4200222             {
4200223                 s +=
4200224                     simaxtoa_fill (value_i,
4200225                                     &string[s], 10,
4200226                                     0, alignment,
4200227                                     filler, remain);
4200228             }
4200229             else
4200230             {
4200231                 s +=
4200232                     imaxtoa_fill (value_i,
4200233                                     &string[s], 10,
4200234                                     0, alignment,
4200235                                     filler, remain);
4200236             }
4200237             f += 3;
4200238         }
4200239     else if (format[f + 2] == 'u')
4200240     {
4200241         // -----
4200242         // unsigned char, base 10.
4200243         value_ui = va_arg (ap, unsigned int);
4200244         s +=
4200245             uimaxtoa_fill (value_ui,
4200246                             &string[s], 10, 0,
4200247                             alignment, filler,
```

```
4200248                                     remain);
4200249             f += 3;
4200250         }
4200251     else if (format[f + 2] == 'o')
4200252     {
4200253         // -----
4200254         // unsigned char, base 8.
4200255         value_ui = va_arg (ap, unsigned int);
4200256         s +=
4200257             uimaxtoa_fill (value_ui,
4200258                           &string[s], 8, 0,
4200259                           alignment, filler,
4200260                           remain);
4200261         f += 3;
4200262     }
4200263     else if (format[f + 2] == 'x')
4200264     {
4200265         // -----
4200266         // unsigned char, base 16.
4200267         value_ui = va_arg (ap, unsigned int);
4200268         s +=
4200269             uimaxtoa_fill (value_ui,
4200270                           &string[s], 16, 0,
4200271                           alignment, filler,
4200272                           remain);
4200273         f += 3;
4200274     }
4200275     else if (format[f + 2] == 'X')
4200276     {
4200277         // -----
4200278         // unsigned char, base 16.
4200279         value_ui = va_arg (ap, unsigned int);
4200280         s +=
4200281             uimaxtoa_fill (value_ui,
4200282                           &string[s], 16, 1,
4200283                           alignment, filler,
4200284                           remain);
```



```
4200322         else
4200323             {
4200324                 s +=
4200325                     imaxtoa_fill (value_i,
4200326                                 &string[s], 10,
4200327                                 0, alignment,
4200328                                 filler, remain);
4200329             }
4200330         f += 2;
4200331     }
4200332     else if (format[f + 1] == 'u')
4200333     {
4200334         // ----- unsigned
4200335         // short int, base 10.
4200336         value_ui = va_arg (ap, unsigned int);
4200337         s +=
4200338             uimaxtoa_fill (value_ui,
4200339                           &string[s], 10, 0,
4200340                           alignment, filler,
4200341                           remain);
4200342         f += 2;
4200343     }
4200344     else if (format[f + 1] == 'o')
4200345     {
4200346         // ----- unsigned
4200347         // short int, base 8.
4200348         value_ui = va_arg (ap, unsigned int);
4200349         s +=
4200350             uimaxtoa_fill (value_ui,
4200351                           &string[s], 8, 0,
4200352                           alignment, filler,
4200353                           remain);
4200354         f += 2;
4200355     }
4200356     else if (format[f + 1] == 'x')
4200357     {
4200358         // ----- unsigned
```



```
4200359         // short int, base 16.
4200360         value_ui = va_arg (ap, unsigned int);
4200361         s +=
4200362             uimaxtoa_fill (value_ui,
4200363                             &string[s], 16, 0,
4200364                             alignment, filler,
4200365                             remain);
4200366         f += 2;
4200367     }
4200368     else if (format[f + 1] == 'X')
4200369     {
4200370         // ----- unsigned
4200371         // short int, base 16.
4200372         value_ui = va_arg (ap, unsigned int);
4200373         s +=
4200374             uimaxtoa_fill (value_ui,
4200375                             &string[s], 16, 1,
4200376                             alignment, filler,
4200377                             remain);
4200378         f += 2;
4200379     }
4200380     else if (format[f + 1] == 'b')
4200381     {
4200382         // ----- unsigned short int,
4200383         // base 2 (extention).
4200384         value_ui = va_arg (ap, unsigned int);
4200385         s +=
4200386             uimaxtoa_fill (value_ui,
4200387                             &string[s], 2, 0,
4200388                             alignment, filler,
4200389                             remain);
4200390         f += 2;
4200391     }
4200392     else
4200393     {
4200394         // ----- unsupported or
4200395         // unknown specifier.
```

```
4200396         f += 1;
4200397     }
4200398 }
4200399 else if (format[f] == 'l' && format[f + 1] != 'l')
4200400 {
4200401     if (format[f + 1] == 'd'
4200402         || format[f + 1] == 'i')
4200403     {
4200404         // -----
4200405         // long int base 10.
4200406         value_i = va_arg (ap, long int);
4200407         if (flag_plus)
4200408             {
4200409                 s +=
4200410                     simaxtoa_fill (value_i,
4200411                                     &string[s], 10,
4200412                                     0, alignment,
4200413                                     filler, remain);
4200414             }
4200415         else
4200416             {
4200417                 s +=
4200418                     imaxtoa_fill (value_i,
4200419                                     &string[s], 10,
4200420                                     0, alignment,
4200421                                     filler, remain);
4200422             }
4200423         f += 2;
4200424     }
4200425     else if (format[f + 1] == 'u')
4200426     {
4200427         // ----- Unsigned
4200428         // long int base 10.
4200429         value_ui = va_arg (ap, unsigned long int);
4200430         s +=
4200431             uimaxtoa_fill (value_ui,
4200432                             &string[s], 10, 0,
```

```
4200433                                     alignment, filler,
4200434                                     remain);
4200435         f += 2;
4200436     }
4200437     else if (format[f + 1] == 'o')
4200438     {
4200439         // ----- Unsigned
4200440         // long int base 8.
4200441         value_ui = va_arg (ap, unsigned long int);
4200442         s +=
4200443             uimaxtoa_fill (value_ui,
4200444                             &string[s], 8, 0,
4200445                             alignment, filler,
4200446                             remain);
4200447         f += 2;
4200448     }
4200449     else if (format[f + 1] == 'x')
4200450     {
4200451         // ----- Unsigned
4200452         // long int base 16.
4200453         value_ui = va_arg (ap, unsigned long int);
4200454         s +=
4200455             uimaxtoa_fill (value_ui,
4200456                             &string[s], 16, 0,
4200457                             alignment, filler,
4200458                             remain);
4200459         f += 2;
4200460     }
4200461     else if (format[f + 1] == 'X')
4200462     {
4200463         // ----- Unsigned
4200464         // long int base 16.
4200465         value_ui = va_arg (ap, unsigned long int);
4200466         s +=
4200467             uimaxtoa_fill (value_ui,
4200468                             &string[s], 16, 1,
4200469                             alignment, filler,
```

```
4200470                                     remain);
4200471             f += 2;
4200472         }
4200473     else if (format[f + 1] == 'b')
4200474     {
4200475         // ----- Unsigned long int
4200476         // base 2 (extention).
4200477         value_ui = va_arg (ap, unsigned long int);
4200478         s +=
4200479             uimaxtoa_fill (value_ui,
4200480                           &string[s], 2, 0,
4200481                           alignment, filler,
4200482                           remain);
4200483         f += 2;
4200484     }
4200485     else
4200486     {
4200487         // ----- unsupported or
4200488         // unknown specifier.
4200489         f += 1;
4200490     }
4200491 }
4200492 else if (format[f] == 'l' && format[f + 1] == 'l')
4200493 {
4200494     if (format[f + 2] == 'd'
4200495         || format[f + 2] == 'i')
4200496     {
4200497         // -----
4200498         // long int base 10.
4200499         value_i = va_arg (ap, long long int);
4200500         if (flag_plus)
4200501         {
4200502             s +=
4200503                 simaxtoa_fill (value_i,
4200504                               &string[s], 10,
4200505                               0, alignment,
4200506                               filler, remain);
```



```
4200544     else if (format[f + 2] == 'x')
4200545     {
4200546         // ----- Unsigned
4200547         // long int base 16.
4200548         value_ui =
4200549             va_arg (ap, unsigned long long int);
4200550         s +=
4200551             uimaxtoa_fill (value_ui,
4200552                           &string[s], 16, 0,
4200553                           alignment, filler,
4200554                           remain);
4200555         f += 3;
4200556     }
4200557     else if (format[f + 2] == 'X')
4200558     {
4200559         // ----- Unsigned
4200560         // long int base 16.
4200561         value_ui =
4200562             va_arg (ap, unsigned long long int);
4200563         s +=
4200564             uimaxtoa_fill (value_ui,
4200565                           &string[s], 16, 1,
4200566                           alignment, filler,
4200567                           remain);
4200568         f += 3;
4200569     }
4200570     else if (format[f + 2] == 'b')
4200571     {
4200572         // ----- Unsigned long int
4200573         // base 2 (extention).
4200574         value_ui =
4200575             va_arg (ap, unsigned long long int);
4200576         s +=
4200577             uimaxtoa_fill (value_ui,
4200578                           &string[s], 2, 0,
4200579                           alignment, filler,
4200580                           remain);
```

```
4200581         f += 3;
4200582     }
4200583     else
4200584     {
4200585         // ----- unsupported or
4200586         // unknown specifier.
4200587         f += 2;
4200588     }
4200589 }
4200590 else if (format[f] == 'j')
4200591 {
4200592     if (format[f + 1] == 'd'
4200593         || format[f + 1] == 'i')
4200594     {
4200595         // -----
4200596         // intmax_t base 10.
4200597         value_i = va_arg (ap, intmax_t);
4200598         if (flag_plus)
4200599             {
4200600                 s +=
4200601                     simaxtoa_fill (value_i,
4200602                                     &string[s], 10,
4200603                                     0, alignment,
4200604                                     filler, remain);
4200605             }
4200606         else
4200607             {
4200608                 s +=
4200609                     imaxtoa_fill (value_i,
4200610                                    &string[s], 10,
4200611                                    0, alignment,
4200612                                    filler, remain);
4200613             }
4200614         f += 2;
4200615     }
4200616     else if (format[f + 1] == 'u')
4200617     {
```

```
4200618 // -----
4200619 // uintmax_t base 10.
4200620 value_ui = va_arg (ap, uintmax_t);
4200621 s +=
4200622     uimaxtoa_fill (value_ui,
4200623                   &string[s], 10, 0,
4200624                   alignment, filler,
4200625                   remain);
4200626     f += 2;
4200627 }
4200628 else if (format[f + 1] == 'o')
4200629 {
4200630     // -----
4200631     // uintmax_t base 8.
4200632     value_ui = va_arg (ap, uintmax_t);
4200633     s +=
4200634         uimaxtoa_fill (value_ui,
4200635                       &string[s], 8, 0,
4200636                       alignment, filler,
4200637                       remain);
4200638     f += 2;
4200639 }
4200640 else if (format[f + 1] == 'x')
4200641 {
4200642     // -----
4200643     // uintmax_t base 16.
4200644     value_ui = va_arg (ap, uintmax_t);
4200645     s +=
4200646         uimaxtoa_fill (value_ui,
4200647                       &string[s], 16, 0,
4200648                       alignment, filler,
4200649                       remain);
4200650     f += 2;
4200651 }
4200652 else if (format[f + 1] == 'X')
4200653 {
4200654     // -----
```



```
4200655         // uintmax_t base 16.
4200656         value_ui = va_arg (ap, uintmax_t);
4200657         s +=
4200658             uimaxtoa_fill (value_ui,
4200659                             &string[s], 16, 1,
4200660                             alignment, filler,
4200661                             remain);
4200662         f += 2;
4200663     }
4200664     else if (format[f + 1] == 'b')
4200665     {
4200666         // ----- uintmax_t
4200667         // base 2 (extention).
4200668         value_ui = va_arg (ap, uintmax_t);
4200669         s +=
4200670             uimaxtoa_fill (value_ui,
4200671                             &string[s], 2, 0,
4200672                             alignment, filler,
4200673                             remain);
4200674         f += 2;
4200675     }
4200676     else
4200677     {
4200678         // ----- unsupported or
4200679         // unknown specifier.
4200680         f += 1;
4200681     }
4200682 }
4200683 else if (format[f] == 'z')
4200684 {
4200685     if (format[f + 1] == 'd'
4200686         || format[f + 1] == 'i'
4200687         || format[f + 1] == 'i')
4200688     {
4200689         // ----- size_t base 10.
4200690         value_ui = va_arg (ap, unsigned long int);
4200691         s +=
```

```
4200692         uimaxtoa_fill (value_ui,
4200693                     &string[s], 10, 0,
4200694                     alignment, filler,
4200695                     remain);
4200696         f += 2;
4200697     }
4200698     else if (format[f + 1] == 'o')
4200699     {
4200700         // ----- size_t base 8.
4200701         value_ui = va_arg (ap, unsigned long int);
4200702         s +=
4200703             uimaxtoa_fill (value_ui,
4200704                           &string[s], 8, 0,
4200705                           alignment, filler,
4200706                           remain);
4200707         f += 2;
4200708     }
4200709     else if (format[f + 1] == 'x')
4200710     {
4200711         // ----- size_t base 16.
4200712         value_ui = va_arg (ap, unsigned long int);
4200713         s +=
4200714             uimaxtoa_fill (value_ui,
4200715                           &string[s], 16, 0,
4200716                           alignment, filler,
4200717                           remain);
4200718         f += 2;
4200719     }
4200720     else if (format[f + 1] == 'X')
4200721     {
4200722         // ----- size_t base 16.
4200723         value_ui = va_arg (ap, unsigned long int);
4200724         s +=
4200725             uimaxtoa_fill (value_ui,
4200726                           &string[s], 16, 1,
4200727                           alignment, filler,
4200728                           remain);
```



```
4200766         else
4200767             {
4200768                 s +=
4200769                     imaxtoa_fill (value_i,
4200770                                 &string[s], 10,
4200771                                 0, alignment,
4200772                                 filler, remain);
4200773             }
4200774             f += 2;
4200775         }
4200776     else if (format[f + 1] == 'u')
4200777     {
4200778         // ----- ptrdiff_t base
4200779         // 10, without sign.
4200780         value_ui = va_arg (ap, unsigned long int);
4200781         s +=
4200782             uimaxtoa_fill (value_ui,
4200783                           &string[s], 10, 0,
4200784                           alignment, filler,
4200785                           remain);
4200786         f += 2;
4200787     }
4200788     else if (format[f + 1] == 'o')
4200789     {
4200790         // ----- ptrdiff_t base
4200791         // 8, without sign.
4200792         value_ui = va_arg (ap, unsigned long int);
4200793         s +=
4200794             uimaxtoa_fill (value_ui,
4200795                           &string[s], 8, 0,
4200796                           alignment, filler,
4200797                           remain);
4200798         f += 2;
4200799     }
4200800     else if (format[f + 1] == 'x')
4200801     {
4200802         // ----- ptrdiff_t base
```

```
4200803         // 16, without sign.
4200804         value_ui = va_arg (ap, unsigned long int);
4200805         s +=
4200806             uimaxtoa_fill (value_ui,
4200807                             &string[s], 16, 0,
4200808                             alignment, filler,
4200809                             remain);
4200810         f += 2;
4200811     }
4200812     else if (format[f + 1] == 'X')
4200813     {
4200814         // ----- ptrdiff_t base
4200815         // 16, without sign.
4200816         value_ui = va_arg (ap, unsigned long int);
4200817         s +=
4200818             uimaxtoa_fill (value_ui,
4200819                             &string[s], 16, 1,
4200820                             alignment, filler,
4200821                             remain);
4200822         f += 2;
4200823     }
4200824     else if (format[f + 1] == 'b')
4200825     {
4200826         // ----- ptrdiff_t base 2, without
4200827         // sign (extention).
4200828         value_ui = va_arg (ap, unsigned long int);
4200829         s +=
4200830             uimaxtoa_fill (value_ui,
4200831                             &string[s], 2, 0,
4200832                             alignment, filler,
4200833                             remain);
4200834         f += 2;
4200835     }
4200836     else
4200837     {
4200838         // ----- unsupported or
4200839         // unknown specifier.
```

```
4200840         f += 1;
4200841     }
4200842 }
4200843 if (format[f] == 'd' || format[f] == 'i')
4200844 {
4200845     // ----- int base 10.
4200846     value_i = va_arg (ap, int);
4200847     if (flag_plus)
4200848     {
4200849         s +=
4200850             simaxtoa_fill (value_i, &string[s],
4200851                             10, 0, alignment,
4200852                             filler, remain);
4200853     }
4200854     else
4200855     {
4200856         s +=
4200857             imaxtoa_fill (value_i, &string[s],
4200858                             10, 0, alignment,
4200859                             filler, remain);
4200860     }
4200861     f += 1;
4200862 }
4200863 else if (format[f] == 'u')
4200864 {
4200865     // -----
4200866     // unsigned int base 10.
4200867     value_ui = va_arg (ap, unsigned int);
4200868     s +=
4200869         uimaxtoa_fill (value_ui, &string[s],
4200870                         10, 0, alignment,
4200871                         filler, remain);
4200872     f += 1;
4200873 }
4200874 else if (format[f] == 'o')
4200875 {
4200876     // ----- unsigned int base 8.
```

```
4200877     value_ui = va_arg (ap, unsigned int);
4200878     s +=
4200879         uimaxtoa_fill (value_ui, &string[s], 8,
4200880                        0, alignment, filler,
4200881                        remain);
4200882     f += 1;
4200883 }
4200884 else if (format[f] == 'x')
4200885 {
4200886     // -----
4200887     // unsigned int base 16.
4200888     value_ui = va_arg (ap, unsigned int);
4200889     s +=
4200890         uimaxtoa_fill (value_ui, &string[s],
4200891                        16, 0, alignment,
4200892                        filler, remain);
4200893     f += 1;
4200894 }
4200895 else if (format[f] == 'X')
4200896 {
4200897     // -----
4200898     // unsigned int base 16.
4200899     value_ui = va_arg (ap, unsigned int);
4200900     s +=
4200901         uimaxtoa_fill (value_ui, &string[s],
4200902                        16, 1, alignment,
4200903                        filler, remain);
4200904     f += 1;
4200905 }
4200906 else if (format[f] == 'b')
4200907 {
4200908     // ----- unsigned int
4200909     // base 2 (extention).
4200910     value_ui = va_arg (ap, unsigned int);
4200911     s +=
4200912         uimaxtoa_fill (value_ui, &string[s], 2,
4200913                        0, alignment, filler,
```

```
4200914                                     remain);
4200915         f += 1;
4200916     }
4200917     else if (format[f] == 'c')
4200918     {
4200919         // ----- unsigned char.
4200920         value_ui = va_arg (ap, unsigned int);
4200921         string[s] = (char) value_ui;
4200922         s += 1;
4200923         f += 1;
4200924     }
4200925     else if (format[f] == 's')
4200926     {
4200927         // ----- string.
4200928         value_cp = va_arg (ap, char *);
4200929         filler = ' ';
4200930
4200931         s +=
4200932             strtostr_fill (value_cp, &string[s],
4200933                           alignment, filler, remain);
4200934         f += 1;
4200935     }
4200936     else
4200937     {
4200938         // ----- unsupported or
4200939         // unknown specifier.
4200940         ;
4200941     }
4200942     // -----
4200943     // End of specifier.
4200944     // -----
4200945     width_string[0] = '\\0';
4200946     precision_string[0] = '\\0';
4200947
4200948     specifier = 0;
4200949     specifier_flags = 0;
4200950     specifier_width = 0;
```



```
4200951     specifier_precision = 0;
4200952     specifier_type = 0;
4200953
4200954     flag_plus = 0;
4200955     flag_minus = 0;
4200956     flag_space = 0;
4200957     flag_alternate = 0;
4200958     flag_zero = 0;
4200959     }
4200960 }
4200961 string[s] = '\0';
4200962 return s;
4200963 }
4200964
4200965 //-----
4200966 // Static functions.
4200967 //-----
4200968 static size_t
4200969 uimaxtoa (uintmax_t integer, char *buffer, int base,
4200970           int uppercase, size_t size)
4200971 {
4200972     // -----
4200973     // Convert a maximum rank integer into a string.
4200974     // -----
4200975
4200976     uintmax_t integer_copy = integer;
4200977     size_t digits;
4200978     int b;
4200979     unsigned char remainder;
4200980
4200981     for (digits = 0; integer_copy > 0; digits++)
4200982     {
4200983         integer_copy = integer_copy / base;
4200984     }
4200985
4200986     if (buffer == NULL && integer == 0)
4200987         return 1;
```

```
4200988     if (buffer == NULL && integer > 0)
4200989         return digits;
4200990
4200991     if (integer == 0)
4200992     {
4200993         buffer[0] = '0';
4200994         buffer[1] = '\\0';
4200995         return 1;
4200996     }
4200997     //
4200998     // Fix the maximum number of digits.
4200999     //
4201000     if (size > 0 && digits > size)
4201001         digits = size;
4201002     //
4201003     *(buffer + digits) = '\\0';    // End of string.
4201004
4201005     for (b = digits - 1; integer != 0 && b >= 0; b--)
4201006     {
4201007         remainder = integer % base;
4201008         integer = integer / base;
4201009
4201010         if (remainder <= 9)
4201011         {
4201012             *(buffer + b) = remainder + '0';
4201013         }
4201014         else
4201015         {
4201016             if (uppercase)
4201017             {
4201018                 *(buffer + b) = remainder - 10 + 'A';
4201019             }
4201020             else
4201021             {
4201022                 *(buffer + b) = remainder - 10 + 'a';
4201023             }
4201024         }
    }
```

```
4201025     }
4201026     return digits;
4201027 }
4201028
4201029 //-----
4201030 static size_t
4201031 imaxtoa (intmax_t integer, char *buffer, int base,
4201032         int uppercase, size_t size)
4201033 {
4201034     // -----
4201035     // Convert a maximum rank integer with sign into a
4201036     // string.
4201037     // -----
4201038
4201039     if (integer >= 0)
4201040     {
4201041         return uimaxtoa (integer, buffer, base,
4201042                         uppercase, size);
4201043     }
4201044     //
4201045     // At this point, there is a negative number, less
4201046     // than zero.
4201047     //
4201048     if (buffer == NULL)
4201049     {
4201050         return uimaxtoa (-integer, NULL, base, uppercase,
4201051                         size) + 1;
4201052     }
4201053
4201054     *buffer = '-';           // The minus sign is needed at
4201055     // the beginning.
4201056     if (size == 1)
4201057     {
4201058         *(buffer + 1) = '\\0';
4201059         return 1;
4201060     }
4201061     else
```

```
4201062     {
4201063         return uimaxtoa (-integer, buffer + 1, base,
4201064                         uppercase, size - 1) + 1;
4201065     }
4201066 }
4201067
4201068 //-----
4201069 static size_t
4201070 simaxtoa (intmax_t integer, char *buffer, int base,
4201071          int uppercase, size_t size)
4201072 {
4201073     // -----
4201074     // Convert a maximum rank integer with sign into a
4201075     // string, placing
4201076     // the sign also if it is positive.
4201077     // -----
4201078
4201079     if (buffer == NULL && integer >= 0)
4201080     {
4201081         return uimaxtoa (integer, NULL, base, uppercase,
4201082                         size) + 1;
4201083     }
4201084
4201085     if (buffer == NULL && integer < 0)
4201086     {
4201087         return uimaxtoa (-integer, NULL, base, uppercase,
4201088                         size) + 1;
4201089     }
4201090     //
4201091     // At this point, 'buffer' is different from NULL.
4201092     //
4201093     if (integer >= 0)
4201094     {
4201095         *buffer = '+';
4201096     }
4201097     else
4201098     {
```

```
420109      *buffer = '-';
420110    }
420111
420112    if (size == 1)
420113    {
420114        *(buffer + 1) = '\\0';
420115        return 1;
420116    }
420117
420118    if (integer >= 0)
420119    {
420120        return uimaxtoa (integer, buffer + 1, base,
420121                        uppercase, size - 1) + 1;
420122    }
420123    else
420124    {
420125        return uimaxtoa (-integer, buffer + 1, base,
420126                        uppercase, size - 1) + 1;
420127    }
420128 }
420129
420130 //-----
420131 static size_t
420132 uimaxtoa_fill (uintmax_t integer, char *buffer,
420133               int base, int uppercase, int width,
420134               int filler, int max)
420135 {
420136     // -----
420137     // Convert a maximum rank integer without sign into
420138     // a string,
420139     // takeing care of the alignment.
420140     // -----
420141
420142     size_t size_i;
420143     size_t size_f;
420144
420145     if (max < 0)
```

```
4201136     return 0;    // «max» deve essere un valore
4201137 // positivo.
4201138
4201139     size_i = uimaxtoa (integer, NULL, base, uppercase, 0);
4201140
4201141     if (width > 0 && max > 0 && width > max)
4201142         width = max;
4201143     if (width < 0 && -max < 0 && width < -max)
4201144         width = -max;
4201145
4201146     if (size_i > abs (width))
4201147     {
4201148         return uimaxtoa (integer, buffer, base,
4201149             uppercase, abs (width));
4201150     }
4201151
4201152     if (width == 0 && max > 0)
4201153     {
4201154         return uimaxtoa (integer, buffer, base,
4201155             uppercase, max);
4201156     }
4201157
4201158     if (width == 0)
4201159     {
4201160         return uimaxtoa (integer, buffer, base,
4201161             uppercase, abs (width));
4201162     }
4201163 //
4201164 // size_i <= abs (width).
4201165 //
4201166     size_f = abs (width) - size_i;
4201167
4201168     if (width < 0)
4201169     {
4201170         // Left alignment.
4201171         uimaxtoa (integer, buffer, base, uppercase, 0);
4201172         memset (buffer + size_i, filler, size_f);
```

```
4201173     }
4201174     else
4201175     {
4201176         // Right alignment.
4201177         memset (buffer, filler, size_f);
4201178         uimaxtoa (integer, buffer + size_f, base,
4201179                 uppercase, 0);
4201180     }
4201181     *(buffer + abs (width)) = '\\0';
4201182
4201183     return abs (width);
4201184 }
4201185
4201186 //-----
4201187 static size_t
4201188 imaxtoa_fill (intmax_t integer, char *buffer, int base,
4201189              int uppercase, int width, int filler, int max)
4201190 {
4201191     // -----
4201192     // Convert a maximum rank integer with sign into a
4201193     // string,
4201194     // takeing care of the alignment.
4201195     // -----
4201196
4201197     size_t size_i;
4201198     size_t size_f;
4201199
4201200     if (max < 0)
4201201         return 0;    // 'max' must be a positive value.
4201202
4201203     size_i = imaxtoa (integer, NULL, base, uppercase, 0);
4201204
4201205     if (width > 0 && max > 0 && width > max)
4201206         width = max;
4201207     if (width < 0 && -max < 0 && width < -max)
4201208         width = -max;
4201209
```

```
4201210     if (size_i > abs (width))
4201211     {
4201212         return imaxtoa (integer, buffer, base, uppercase,
4201213                         abs (width));
4201214     }
4201215
4201216     if (width == 0 && max > 0)
4201217     {
4201218         return imaxtoa (integer, buffer, base, uppercase,
4201219                         max);
4201220     }
4201221
4201222     if (width == 0)
4201223     {
4201224         return imaxtoa (integer, buffer, base, uppercase,
4201225                         abs (width));
4201226     }
4201227
4201228     // size_i <= abs (width).
4201229
4201230     size_f = abs (width) - size_i;
4201231
4201232     if (width < 0)
4201233     {
4201234         // Left alignment.
4201235         imaxtoa (integer, buffer, base, uppercase, 0);
4201236         memset (buffer + size_i, filler, size_f);
4201237     }
4201238     else
4201239     {
4201240         // Right alignment.
4201241         memset (buffer, filler, size_f);
4201242         imaxtoa (integer, buffer + size_f, base,
4201243                 uppercase, 0);
4201244     }
4201245     *(buffer + abs (width)) = '\\0';
4201246
```



```
4201247     return abs (width);
4201248 }
4201249
4201250 //-----
4201251 static size_t
4201252 simaxtoa_fill (intmax_t integer, char *buffer,
4201253               int base, int uppercase, int width,
4201254               int filler, int max)
4201255 {
4201256     // -----
4201257     // Convert a maximum rank integer with sign into a
4201258     // string,
4201259     // placing the sign also if it is positive and
4201260     // takeing care of the
4201261     // alignment.
4201262     // -----
4201263
4201264     size_t size_i;
4201265     size_t size_f;
4201266
4201267     if (max < 0)
4201268         return 0;    // 'max' must be a positive value.
4201269
4201270     size_i = simaxtoa (integer, NULL, base, uppercase, 0);
4201271
4201272     if (width > 0 && max > 0 && width > max)
4201273         width = max;
4201274     if (width < 0 && -max < 0 && width < -max)
4201275         width = -max;
4201276
4201277     if (size_i > abs (width))
4201278     {
4201279         return simaxtoa (integer, buffer, base,
4201280                         uppercase, abs (width));
4201281     }
4201282
4201283     if (width == 0 && max > 0)
```

```
4201284     {
4201285         return simaxtoa (integer, buffer, base,
4201286                         uppercase, max);
4201287     }
4201288
4201289     if (width == 0)
4201290     {
4201291         return simaxtoa (integer, buffer, base,
4201292                         uppercase, abs (width));
4201293     }
4201294     //
4201295     // size_i <= abs (width).
4201296     //
4201297     size_f = abs (width) - size_i;
4201298
4201299     if (width < 0)
4201300     {
4201301         // Left alignment.
4201302         simaxtoa (integer, buffer, base, uppercase, 0);
4201303         memset (buffer + size_i, filler, size_f);
4201304     }
4201305     else
4201306     {
4201307         // Right alignment.
4201308         memset (buffer, filler, size_f);
4201309         simaxtoa (integer, buffer + size_f, base,
4201310                 uppercase, 0);
4201311     }
4201312     *(buffer + abs (width)) = '\\0';
4201313
4201314     return abs (width);
4201315 }
4201316
4201317 //-----
4201318 static size_t
4201319 strtostr_fill (char *string, char *buffer, int width,
4201320               int filler, int max)
```

```
4201321 {
4201322     // -----
4201323     // Transfer a string with care for the alignment.
4201324     // -----
4201325
4201326     size_t size_s;
4201327     size_t size_f;
4201328
4201329     if (max < 0)
4201330         return 0;    // 'max' must be a positive value.
4201331
4201332     size_s = strlen (string);
4201333
4201334     if (width > 0 && max > 0 && width > max)
4201335         width = max;
4201336     if (width < 0 && -max < 0 && width < -max)
4201337         width = -max;
4201338
4201339     if (width != 0 && size_s > abs (width))
4201340     {
4201341         memcpy (buffer, string, abs (width));
4201342         buffer[width] = '\0';
4201343         return width;
4201344     }
4201345
4201346     if (width == 0 && max > 0 && size_s > max)
4201347     {
4201348         memcpy (buffer, string, max);
4201349         buffer[max] = '\0';
4201350         return max;
4201351     }
4201352
4201353     if (width == 0 && max > 0 && size_s < max)
4201354     {
4201355         memcpy (buffer, string, size_s);
4201356         buffer[size_s] = '\0';
4201357         return size_s;

```

```
4201358     }
4201359     //
4201360     // width != 0
4201361     // size_s <= abs (width)
4201362     //
4201363     size_f = abs (width) - size_s;
4201364
4201365     if (width < 0)
4201366     {
4201367         // Right alignment.
4201368         memset (buffer, filler, size_f);
4201369         strncpy (buffer + size_f, string, size_s);
4201370     }
4201371     else
4201372     {
4201373         // Left alignment.
4201374         strncpy (buffer, string, size_s);
4201375         memset (buffer + size_s, filler, size_f);
4201376     }
4201377     *(buffer + abs (width)) = '\0';
4201378
4201379     return abs (width);
4201380 }
```

95.18.43 lib/stdio/vsprintf.c

«

Si veda la sezione [88.137](#).

```
4210001 #include <stdio.h>
4210002 //-----
4210003 int
4210004 vsprintf (char *restrict string,
4210005           const char *restrict format, va_list arg)
4210006 {
4210007     return (vsnprintf (string, BUFSIZ, format, arg));
4210008 }
```

95.18.44 lib/stdio/vsscanf.c



Si veda la sezione [88.138](#).

```
4220001 #include <stdio.h>
4220002
4220003 //-----
4220004 int vfsscanf (FILE * restrict fp, const char *string,
4220005              const char *restrict format, va_list ap);
4220006 //-----
4220007 int
4220008 vsscanf (const char *string,
4220009          const char *restrict format, va_list ap)
4220010 {
4220011     return (vfsscanf (NULL, string, format, ap));
4220012 }
4220013
4220014 //-----
```

95.19 os32: «lib/stdlib.h»



Si veda la sezione [91.3](#).

```
4230001 #ifndef _STDLIB_H
4230002 #define _STDLIB_H      1
4230003 //-----
4230004 #include <size_t.h>
4230005 #include <wchar_t.h>
4230006 #include <NULL.h>
4230007 #include <limits.h>
4230008 #include <restrict.h>
4230009 #include <stdint.h>
4230010 //-----
4230011 typedef struct
4230012 {
4230013     int quot;
4230014     int rem;
4230015 } div_t;
```

```
4230016 //-----
4230017 typedef struct
4230018 {
4230019     long int quot;
4230020     long int rem;
4230021 } ldiv_t;
4230022 //-----
4230023 typedef struct
4230024 {
4230025     long long int quot;
4230026     long long int rem;
4230027 } lldiv_t;
4230028 //-----
4230029 typedef void (*atexit_t) (void);          // Non standard.
4230030                                         // [1]
4230031 //
4230032 // [1] The type 'atexit_t' is a pointer to a function
4230033 //       for the "at exit" procedure, with no parameters
4230034 //       and returning void. With the declaration of type
4230035 //       'atexit_t', the function prototype of 'atexit()'
4230036 //       is easier to declare and to understand. Original
4230037 //       declaration is:
4230038 //
4230039 //       int atexit (void (*function) (void));
4230040 //
4230041 //-----
4230042 typedef struct
4230043 {
4230044     uintptr_t allocated:1, filler:1, next:30;
4230045 } _alloc_head_t;          // Non standard [2]
4230046 //
4230047 // [2] This is used for the 'malloc()' management, as
4230048 //       the pointer to the following element of memory,
4230049 //       that might be free or allocated.
4230050 //
4230051 // La dimensione di «uintptr_t» condiziona la struttura
4230052 // «mm_head_t» e la dimensione delle unità minime di
```

```
4230053 // memoria allocata. «uintptr_t» è da 32 bit, così
4230054 // l'immagine del kernel è allineata a blocchi da
4230055 // 32 bit e così deve essere anche per gli altri
4230056 // blocchi di memoria.
4230057 // Essendo i blocchi di memoria multipli di 32 bit, gli
4230058 // indirizzi sono sempre multipli di 4 (4 byte);
4230059 // pertanto, servono solo 30 bit per rappresentare
4230060 // l'indirizzo, che poi viene ottenuto moltiplicandolo
4230061 // per quattro. Di conseguenza, il bit meno
4230062 // significativo viene usato per annotare se il blocco
4230063 // di memoria è libero e il bit successivo non viene
4230064 // usato. Questo meccanismo potrebbe essere usato anche
4230065 // con un indirizzamento a 16 bit, dove servirebbero 15
4230066 // bit per indirizzi multipli di due byte.
4230067 //
4230068 //-----
4230069 #define EXIT_FAILURE      1
4230070 #define EXIT_SUCCESS     0
4230071 #define RAND_MAX         INT_MAX
4230072 #define MB_CUR_MAX      ((size_t) MB_LEN_MAX)
4230073 //-----
4230074 void _Exit (int status);
4230075 void abort (void);
4230076 int abs (int j);
4230077 int atexit (atexit_t function);
4230078 int atoi (const char *string);
4230079 long int atol (const char *string);
4230080 #define calloc(b, s) (malloc ((b) * (s)))
4230081 div_t div (int numer, int denom);
4230082 void exit (int status);
4230083 void free (void *ptr);
4230084 char *getenv (const char *name);
4230085 long int labs (long int j);
4230086 long long int llabs (long long int j);
4230087 ldiv_t ldiv (long int numer, long int denom);
4230088 lldiv_t lldiv (long long int numer, long long int denom);
4230089 void *malloc (size_t size);
```

```

4230090 int putenv (const char *string);
4230091 void qsort (void *base, size_t nmem, size_t size,
4230092             int (*compare) (const void *, const void *));
4230093 int rand (void);
4230094 void *realloc (void *ptr, size_t size);
4230095 int setenv (const char *name, const char *value,
4230096             int overwrite);
4230097 void srand (unsigned int seed);
4230098 long int strtol (const char *restrict string,
4230099                 char **restrict endptr, int base);
4230100 unsigned long int strtoul (const char *restrict string,
4230101                             char **restrict endptr,
4230102                             int base);
4230103 //int system (const char *string);
4230104 int unsetenv (const char *name);
4230105 //-----
4230106 #endif

```

95.19.1	lib/stdlib/_Exit.c	2017
95.19.2	lib/stdlib/abort.c	2018
95.19.3	lib/stdlib/abs.c	2019
95.19.4	lib/stdlib/atexit.c	2020
95.19.5	lib/stdlib/atoi.c	2021
95.19.6	lib/stdlib/atol.c	2022
95.19.7	lib/stdlib/div.c	2023
95.19.8	lib/stdlib/environment.c	2024
95.19.9	lib/stdlib/exit.c	2026
95.19.10	lib/stdlib/getenv.c	2027
95.19.11	lib/stdlib/labs.c	2029

95.19.12	lib/stdlib/ldiv.c	2030
95.19.13	lib/stdlib/llabs.c	2030
95.19.14	lib/stdlib/lldiv.c	2031
95.19.15	lib/stdlib/putenv.c	2031
95.19.16	lib/stdlib/qsort.c	2034
95.19.17	lib/stdlib/rand.c	2038
95.19.18	lib/stdlib/setenv.c	2039
95.19.19	lib/stdlib/strtol.c	2043
95.19.20	lib/stdlib/strtoul.c	2049
95.19.21	lib/stdlib/unsetenv.c	2049
95.19.22	lib/stdlib_alloc/_alloc_list.c	2052
95.19.23	lib/stdlib_alloc/free.c	2054
95.19.24	lib/stdlib_alloc/malloc.c	2056
95.19.25	lib/stdlib_alloc/realloc.c	2063

95.19.1 lib/stdlib/_Exit.c

Si veda la sezione [87.2](#).

```
4240001 #include <stdlib.h>
4240002 #include <sys/os32.h>
4240003 //-----
4240004 void
4240005 _Exit (int status)
4240006 {
4240007     sysmsg_exit_t msg;
4240008     //
4240009     // Only the low eight bit are returned.
```

```
4240010 //
4240011 msg.status = (status & 0xFF);
4240012 //
4240013 //
4240014 //
4240015 sys (SYS_EXIT, &msg, (sizeof msg));
4240016 //
4240017 // Should not return from system call, but if it
4240018 // does, loop
4240019 // forever:
4240020 //
4240021 while (1);
4240022 }
```

95.19.2 lib/stdlib/abort.c



Si veda la sezione [88.2](#).

```
4250001 #include <stdlib.h>
4250002 #include <sys/types.h>
4250003 #include <signal.h>
4250004 #include <unistd.h>
4250005 //-----
4250006 void
4250007 abort (void)
4250008 {
4250009     pid_t pid;
4250010     sighandler_t sig_previous;
4250011     //
4250012     // Set 'SIGABRT' to a default action.
4250013     //
4250014     sig_previous = signal (SIGABRT, SIG_DFL);
4250015     //
4250016     // If the previous action was something different
4250017     // than symbolic
4250018     // ones, configure again the previous action.
4250019     //
```

```
4250020     if (sig_previous != SIG_DFL &&
4250021         sig_previous != SIG_IGN && sig_previous != SIG_ERR)
4250022     {
4250023         signal (SIGABRT, sig_previous);
4250024     }
4250025     //
4250026     // Get current process ID and sent the signal.
4250027     //
4250028     pid = getpid ();
4250029     kill (pid, SIGABRT);
4250030     //
4250031     // Second chance
4250032     //
4250033     for (;;)
4250034     {
4250035         signal (SIGABRT, SIG_DFL);
4250036         pid = getpid ();
4250037         kill (pid, SIGABRT);
4250038     }
4250039 }
```

95.19.3 lib/stdlib/abs.c

Si veda la sezione [88.3](#).

```
4260001 #include <stdlib.h>
4260002 //-----
4260003 int
4260004 abs (int j)
4260005 {
4260006     if (j < 0)
4260007     {
4260008         return -j;
4260009     }
4260010     else
4260011     {
4260012         return j;
```



```
4260013     }
4260014 }
```

95.19.4 lib/stdlib/atexit.c



Si veda la sezione [88.7](#).

```
4270001 #include <stdlib.h>
4270002 //-----
4270003 atexit_t _atexit_table[ATEXTIT_MAX];
4270004 //-----
4270005 void
4270006 _atexit_setup (void)
4270007 {
4270008     int a;
4270009     //
4270010     for (a = 0; a < ATEXTIT_MAX; a++)
4270011     {
4270012         _atexit_table[a] = NULL;
4270013     }
4270014 }
4270015
4270016 //-----
4270017 int
4270018 atexit (atexit_t function)
4270019 {
4270020     int a;
4270021     //
4270022     if (function == NULL)
4270023     {
4270024         return (-1);
4270025     }
4270026     //
4270027     for (a = 0; a < ATEXTIT_MAX; a++)
4270028     {
4270029         if (_atexit_table[a] == NULL)
4270030         {
```

```
4270031         _atexit_table[a] = function;
4270032         return (0);
4270033     }
4270034 }
4270035 //
4270036 return (-1);
4270037 }
```

95.19.5 lib/stdlib/atoi.c



Si veda la sezione [88.8](#).

```
4280001 #include <stdlib.h>
4280002 #include <ctype.h>
4280003 //-----
4280004 int
4280005 atoi (const char *string)
4280006 {
4280007     int i;
4280008     int sign = +1;
4280009     int number;
4280010     //
4280011     for (i = 0; isspace (string[i]); i++)
4280012     {
4280013         ;
4280014     }
4280015     //
4280016     if (string[i] == '+')
4280017     {
4280018         sign = +1;
4280019         i++;
4280020     }
4280021     else if (string[i] == '-')
4280022     {
4280023         sign = -1;
4280024         i++;
4280025     }
```

```
4280026 //
4280027 for (number = 0; isdigit (string[i]); i++)
4280028 {
4280029     number *= 10;
4280030     number += (string[i] - '0');
4280031 }
4280032 //
4280033 number *= sign;
4280034 //
4280035 return number;
4280036 }
```

95.19.6 lib/stdlib/atol.c

«

Si veda la sezione [88.8](#).

```
4290001 #include <stdlib.h>
4290002 #include <ctype.h>
4290003 //-----
4290004 long int
4290005 atol (const char *string)
4290006 {
4290007     int i;
4290008     int sign = +1;
4290009     long int number;
4290010     //
4290011     for (i = 0; isspace (string[i]); i++)
4290012     {
4290013         ;
4290014     }
4290015     //
4290016     if (string[i] == '+')
4290017     {
4290018         sign = +1;
4290019         i++;
4290020     }
4290021     else if (string[i] == '-')
```

```
4290022     {
4290023         sign = -1;
4290024         i++;
4290025     }
4290026     //
4290027     for (number = 0; isdigit (string[i]); i++)
4290028     {
4290029         number *= 10;
4290030         number += (string[i] - '0');
4290031     }
4290032     //
4290033     number *= sign;
4290034     //
4290035     return number;
4290036 }
```

95.19.7 lib/stdlib/div.c

Si veda la sezione [88.17](#).

```
4300001 #include <stdlib.h>
4300002 //-----
4300003 div_t
4300004 div (int numer, int denom)
4300005 {
4300006     div_t d;
4300007     d.quot = numer / denom;
4300008     d.rem = numer % denom;
4300009     return d;
4300010 }
```

95.19.8 lib/stdlib/environment.c



Si veda la sezione 91.1.

```
4310001 #include <stdlib.h>
4310002 #include <string.h>
4310003 //-----
4310004 // This file contains a non standard definition,
4310005 // related to the environment handling.
4310006 //
4310007 // The file 'crt0.s', before calling the main function,
4310008 // calls the function '_environment_setup()', that is
4310009 // responsible for initializing the array
4310010 // '_environment_table[][]' and for copying the content
4310011 // of the environment, as it comes from the 'exec()'
4310012 // system call.
4310013 //
4310014 // The pointers to the environment strings organised
4310015 // inside the array '_environment_table[][]', are also
4310016 // copied inside the array of pointers
4310017 // '_environment[]'.
4310018 //
4310019 // After all that is done, inside 'crt0.s', the pointer
4310020 // to '_environment[]' is copied to the traditional
4310021 // variable 'environ' and also to the previous value of
4310022 // the pointer variable 'envp'.
4310023 //
4310024 // This way, applications will get the environment, but
4310025 // organised inside the table '_environment_table[][]'.
4310026 // So, functions like 'getenv()' and 'setenv()' do know
4310027 // where to look for.
4310028 //
4310029 // It is useful to notice that there is no prototype
4310030 // and no extern declaration inside the file
4310031 // <stdlib.h>, about this function and these arrays,
4310032 // because applications do not have to know about it.
4310033 //
4310034 // Please notice that 'environ' could be just the same
```



```
4310035 // as '_environment' here, but the common use puts
4310036 // 'environ' inside <unistd.h>, although for this
4310037 // implementation it should be better placed inside
4310038 // <stdlib.h>.
4310039 //
4310040 //-----
4310041 char _environment_table[ARG_MAX / 32][ARG_MAX / 16];
4310042 char *_environment[ARG_MAX / 32 + 1];
4310043 //-----
4310044 void
4310045 _environment_setup (char *envp[])
4310046 {
4310047     int e;
4310048     int s;
4310049     //
4310050     // Reset the '_environment_table[][]' array.
4310051     //
4310052     for (e = 0; e < ARG_MAX / 32; e++)
4310053     {
4310054         for (s = 0; s < ARG_MAX / 16; s++)
4310055         {
4310056             _environment_table[e][s] = 0;
4310057         }
4310058     }
4310059     //
4310060     // Set the '_environment[]' pointers. The final
4310061     // extra element must
4310062     // be a NULL pointer.
4310063     //
4310064     for (e = 0; e < ARG_MAX / 32; e++)
4310065     {
4310066         _environment[e] = _environment_table[e];
4310067     }
4310068     _environment[ARG_MAX / 32] = NULL;
4310069     //
4310070     // Copy the environment inside the array, but only
4310071     // if 'envp' is
```

```
4310072 // not NULL.
4310073 //
4310074 if (envp != NULL)
4310075 {
4310076     for (e = 0; envp[e] != NULL && e < ARG_MAX / 32; e++)
4310077     {
4310078         strncpy (_environment_table[e], envp[e],
4310079                 (ARG_MAX / 16) - 1);
4310080     }
4310081 }
4310082 }
```

95.19.9 lib/stdlib/exit.c

«

Si veda la sezione [88.7](#).

```
4320001 #include <stdlib.h>
4320002 #include <stdio.h>
4320003 //-----
4320004 extern atexit_t _atexit_table[];
4320005 //-----
4320006 void
4320007 exit (int status)
4320008 {
4320009     int a;
4320010     //
4320011     // The "at exit" functions must be called in reverse
4320012     // order.
4320013     //
4320014     for (a = (ATEXIT_MAX - 1); a >= 0; a--)
4320015     {
4320016         if (_atexit_table[a] != NULL)
4320017         {
4320018             (*_atexit_table[a]) ();
4320019         }
4320020     }
4320021     //
```

```
4320022 // Now: really exit.
4320023 //
4320024 _Exit (status);
4320025 //
4320026 // Should not return from system call, but if it
4320027 // does, loop
4320028 // forever:
4320029 //
4320030 while (1);
4320031 }
```

95.19.10 lib/stdlib/getenv.c

Si veda la sezione [88.52](#).

```
4330001 #include <stdlib.h>
4330002 #include <string.h>
4330003 //-----
4330004 extern char *_environment[];
4330005 //-----
4330006 char *
4330007 getenv (const char *name)
4330008 {
4330009     int e;           // First index: environment table
4330010     // items.
4330011     int f;           // Second index: environment string
4330012     // scan.
4330013     char *value;    // Pointer to the environment value
4330014     // found.
4330015     //
4330016     // Check if the input is valid. No error is
4330017     // reported.
4330018     //
4330019     if (name == NULL || strlen (name) == 0)
4330020     {
4330021         return (NULL);
4330022     }
```

```
4330023 //
4330024 // Scan the environment table items, with index 'e'.
4330025 // The pointer
4330026 // 'value' is initialized to NULL. If the pointer
4330027 // 'value' gets a
4330028 // valid pointer, the environment variable was found
4330029 // and a
4330030 // pointer to the beginning of its value is
4330031 // available.
4330032 //
4330033 for (value = NULL, e = 0; e < ARG_MAX / 32; e++)
4330034 {
4330035     //
4330036     // Scan the string of the environment item, with
4330037     // index 'f'.
4330038     // The scan continue until 'name[f]' and
4330039     // '_environment[e][f]'
4330040     // are equal.
4330041     //
4330042     for (f = 0;
4330043          f < ARG_MAX / 16 - 1
4330044          && name[f] == _environment[e][f]; f++)
4330045     {
4330046         ; // Just scan.
4330047     }
4330048     //
4330049     // At this point, 'name[f]' and
4330050     // '_environment[e][f]' are
4330051     // different: if 'name[f]' is zero the name
4330052     // string is
4330053     // terminated; if '_environment[e][f]' is also
4330054     // equal to '=',
4330055     // the environment item is corresponding to the
4330056     // requested name.
4330057     //
4330058     if (name[f] == 0 && _environment[e][f] == '=')
4330059     {
```

```
4330060 //
4330061 // The pointer to the beginning of the
4330062 // environment value is
4330063 // calculated, and the external loop exit.
4330064 //
4330065 value = &_environment[e][f + 1];
4330066 break;
4330067 }
4330068 }
4330069 //
4330070 // The 'value' is returned: if it is still NULL,
4330071 // then, no
4330072 // environment variable with the requested name was
4330073 // found.
4330074 //
4330075 return (value);
4330076 }
```

95.19.11 lib/stdlib/labs.c



Si veda la sezione [88.3](#).

```
4340001 #include <stdlib.h>
4340002 //-----
4340003 long int
4340004 labs (long int j)
4340005 {
4340006     if (j < 0)
4340007     {
4340008         return -j;
4340009     }
4340010     else
4340011     {
4340012         return j;
4340013     }
4340014 }
```

95.19.12 lib/stdlib/ldiv.c



Si veda la sezione [88.17](#).

```
4350001 #include <stdlib.h>
4350002 //-----
4350003 ldiv_t
4350004 ldiv (long int numer, long int denom)
4350005 {
4350006     ldiv_t d;
4350007     d.quot = numer / denom;
4350008     d.rem = numer % denom;
4350009     return d;
4350010 }
```

95.19.13 lib/stdlib/llabs.c



Si veda la sezione [88.3](#).

```
4360001 #include <stdlib.h>
4360002 //-----
4360003 long long int
4360004 llabs (long long int j)
4360005 {
4360006     if (j < 0)
4360007     {
4360008         return -j;
4360009     }
4360010     else
4360011     {
4360012         return j;
4360013     }
4360014 }
```

95.19.14 lib/stdlib/lldiv.c



Si veda la sezione [88.17](#).

```
4370001 #include <stdlib.h>
4370002 //-----
4370003 lldiv_t
4370004 lldiv (long long int numer, long long int denom)
4370005 {
4370006     lldiv_t d;
4370007     d.quot = numer / denom;
4370008     d.rem = numer % denom;
4370009     return d;
4370010 }
```

95.19.15 lib/stdlib/putenv.c



Si veda la sezione [88.94](#).

```
4380001 #include <stdlib.h>
4380002 #include <string.h>
4380003 #include <errno.h>
4380004 //-----
4380005 extern char *_environment[];
4380006 //-----
4380007 int
4380008 putenv (const char *string)
4380009 {
4380010     int e;           // First index: environment table
4380011     // items.
4380012     int f;           // Second index: environment string
4380013     // scan.
4380014     //
4380015     // Check if the input is empty. No error is
4380016     // reported.
4380017     //
4380018     if (string == NULL || strlen (string) == 0)
4380019     {
```

```
4380020     return (0);
4380021     }
4380022     //
4380023     // Check if the input is valid: there must be a '='
4380024     // sign.
4380025     // Error here is reported.
4380026     //
4380027     if (strchr (string, '=') == NULL)
4380028     {
4380029         errset (EINVAL); // Invalid argument.
4380030         return (-1);
4380031     }
4380032     //
4380033     // Scan the environment table items, with index 'e'.
4380034     // The intent is
4380035     // to find a previous environment variable with the
4380036     // same name.
4380037     //
4380038     for (e = 0; e < ARG_MAX / 32; e++)
4380039     {
4380040         //
4380041         // Scan the string of the environment item, with
4380042         // index 'f'.
4380043         // The scan continue until 'string[f]' and
4380044         // '_environment[e][f]'
4380045         // are equal.
4380046         //
4380047         for (f = 0;
4380048              f < ARG_MAX / 16 - 1
4380049              && string[f] == _environment[e][f]; f++)
4380050         {
4380051             ; // Just scan.
4380052         }
4380053         //
4380054         // At this point, 'string[f-1]' and
4380055         // '_environment[e][f-1]'
4380056         // should contain '='. If it is so, the
```



```
4380057     // environment is replaced.
4380058     //
4380059     if (string[f - 1] == '='
4380060         && _environment[e][f - 1] == '=')
4380061     {
4380062         //
4380063         // The environment item was found: now
4380064         // replace the pointer.
4380065         //
4380066         _environment[e] = (char *) string;
4380067         //
4380068         // Return.
4380069         //
4380070         return (0);
4380071     }
4380072 }
4380073 //
4380074 // The item was not found. Scan again for a free
4380075 // slot.
4380076 //
4380077 for (e = 0; e < ARG_MAX / 32; e++)
4380078 {
4380079     if (_environment[e] == NULL
4380080         || _environment[e][0] == 0)
4380081     {
4380082         //
4380083         // An empty item was found and the pointer
4380084         // will be
4380085         // replaced.
4380086         //
4380087         _environment[e] = (char *) string;
4380088         //
4380089         // Return.
4380090         //
4380091         return (0);
4380092     }
4380093 }
```

```
4380094 //
4380095 // Sorry: the empty slot was not found!
4380096 //
4380097 errset (ENOMEM); // Not enough space.
4380098 return (-1);
4380099 }
```

95.19.16 lib/stdlib/qsort.c

«

Si veda la sezione [88.96](#).

```
4390001 #include <stdlib.h>
4390002 #include <string.h>
4390003 #include <errno.h>
4390004 //-----
4390005 static int part (char *array, size_t size, int a,
4390006                 int z, int (*compare) (const void *,
4390007                                         const void *));
4390008 static void sort (char *array, size_t size, int a,
4390009                  int z, int (*compare) (const void *,
4390010                                         const void *));
4390011 //-----
4390012 void
4390013 qsort (void *base, size_t nmemb, size_t size,
4390014        int (*compare) (const void *, const void *))
4390015 {
4390016     if (size <= 1)
4390017     {
4390018         //
4390019         // There is nothing to sort!
4390020         //
4390021         return;
4390022     }
4390023     else
4390024     {
4390025         sort ((char *) base, size, 0, (int) (nmemb - 1),
4390026              compare);
```

```
4390027     }
4390028 }
4390029
4390030 //-----
4390031 static void
4390032 sort (char *array, size_t size, int a, int z,
4390033       int (*compare) (const void *, const void *))
4390034 {
4390035     int loc;
4390036     //
4390037     if (z > a)
4390038     {
4390039         loc = part (array, size, a, z, compare);
4390040         if (loc >= 0)
4390041         {
4390042             sort (array, size, a, loc - 1, compare);
4390043             sort (array, size, loc + 1, z, compare);
4390044         }
4390045     }
4390046 }
4390047
4390048 //-----
4390049 static int
4390050 part (char *array, size_t size, int a, int z,
4390051       int (*compare) (const void *, const void *))
4390052 {
4390053     int i;
4390054     int loc;
4390055     char *swap;
4390056     //
4390057     if (z <= a)
4390058     {
4390059         errset (EUNKNOWN);           // Should never
4390060                                         // happen.
4390061         return (-1);
4390062     }
4390063     //
```

```
4390064 // Index 'i' after the first element; index 'loc' at
4390065 // the last
4390066 // position.
4390067 //
4390068 i = a + 1;
4390069 loc = z;
4390070 //
4390071 // Prepare space in memory for element swap.
4390072 //
4390073 swap = malloc (size);
4390074 if (swap == NULL)
4390075     {
4390076         errset (ENOMEM);
4390077         return (-1);
4390078     }
4390079 //
4390080 // Loop as long as index 'loc' is higher than index
4390081 // 'i'.
4390082 // When index 'loc' is less or equal to index 'i',
4390083 // then, index 'loc' is the right position for the
4390084 // first element of the current piece of array.
4390085 //
4390086 for (;;)
4390087     {
4390088         //
4390089         // Index 'i' goes up...
4390090         //
4390091         for (; i < loc; i++)
4390092             {
4390093                 if (compare
4390094                     (&array[i * size], &array[a * size]) > 0)
4390095                     {
4390096                         break;
4390097                     }
4390098             }
4390099         //
4390100         // Index 'loc' gose down...
```

```
4390101 //
4390102 for (;;) loc--
4390103 {
4390104     if (compare
4390105         (&array[loc * size], &array[a * size]) <= 0)
4390106     {
4390107         break;
4390108     }
4390109 }
4390110 //
4390111 // Swap elements related to index 'i' and 'loc'.
4390112 //
4390113 if (loc <= i)
4390114 {
4390115     //
4390116     // The array is completely scanned.
4390117     //
4390118     break;
4390119 }
4390120 else
4390121 {
4390122     memcpy (swap, &array[loc * size], size);
4390123     memcpy (&array[loc * size], &array[i * size],
4390124             size);
4390125     memcpy (&array[i * size], swap, size);
4390126 }
4390127 }
4390128 //
4390129 // Swap the first element with the one related to
4390130 // the
4390131 // index 'loc'.
4390132 //
4390133 memcpy (swap, &array[loc * size], size);
4390134 memcpy (&array[loc * size], &array[a * size], size);
4390135 memcpy (&array[a * size], swap, size);
4390136 //
4390137 // Free the swap memory.
```

```
4390138 //
4390139 free (swap);
4390140 //
4390141 // Return the index 'loc'.
4390142 //
4390143 return (loc);
4390144 }
```

95.19.17 lib/stdlib/rand.c



Si veda la sezione [88.97](#).

```
4400001 #include <stdlib.h>
4400002 //-----
4400003 static unsigned int _srand = 1; // The '_srand' rank
4400004 // must be at least
4400005 // 'unsigned int' and
4400006 // must be able to
4400007 // represent the value
4400008 // 'RAND_MAX'.
4400009 //-----
4400010 int
4400011 rand (void)
4400012 {
4400013     _srand = _srand * 12345 + 123;
4400014     return _srand % ((unsigned int) RAND_MAX + 1);
4400015 }
4400016
4400017 //-----
4400018 void
4400019 srand (unsigned int seed)
4400020 {
4400021     _srand = seed;
4400022 }
```

95.19.18 lib/stdlib/setenv.c



Si veda la sezione [88.104](#).

```
4410001 #include <stdlib.h>
4410002 #include <string.h>
4410003 #include <errno.h>
4410004 //-----
4410005 extern char *_environment[];
4410006 extern char *_environment_table[];
4410007 //-----
4410008 int
4410009 setenv (const char *name, const char *value, int overwrite)
4410010 {
4410011     int e;           // First index: environment table
4410012                    // items.
4410013     int f;           // Second index: environment string
4410014                    // scan.
4410015     //
4410016     // Check if the input is empty. No error is
4410017     // reported.
4410018     //
4410019     if (name == NULL || strlen (name) == 0)
4410020     {
4410021         return (0);
4410022     }
4410023     //
4410024     // Check if the input is valid: error here is
4410025     // reported.
4410026     //
4410027     if (strchr (name, '=') != NULL)
4410028     {
4410029         errset (EINVAL); // Invalid argument.
4410030         return (-1);
4410031     }
4410032     //
4410033     // Check if the input is too big.
4410034     //
```

```
4410035 if ((strlen (name) + strlen (value) + 2) > ARG_MAX / 16)
4410036 {
4410037     //
4410038     // The environment to be saved is bigger than
4410039     // the
4410040     // available string size, inside
4410041     // '_environment_table[]'.
4410042     //
4410043     errset (ENOMEM); // Not enough space.
4410044     return (-1);
4410045 }
4410046 //
4410047 // Scan the environment table items, with index 'e'.
4410048 // The intent is
4410049 // to find a previous environment variable with the
4410050 // same name.
4410051 //
4410052 for (e = 0; e < ARG_MAX / 32; e++)
4410053 {
4410054     //
4410055     // Scan the string of the environment item, with
4410056     // index 'f'.
4410057     // The scan continue until 'name[f]' and
4410058     // '_environment[e][f]'
4410059     // are equal.
4410060     //
4410061     for (f = 0;
4410062          f < ARG_MAX / 16 - 1
4410063          && name[f] == _environment[e][f]; f++)
4410064     {
4410065         ; // Just scan.
4410066     }
4410067     //
4410068     // At this point, 'name[f]' and
4410069     // '_environment[e][f]' are
4410070     // different: if 'name[f]' is zero the name
4410071     // string is
```



```
4410072 // terminated; if '_environment[e][f]' is also
4410073 // equal to '=',
4410074 // the environment item is corresponding to the
4410075 // requested name.
4410076 //
4410077 if (name[f] == 0 && _environment[e][f] == '=')
4410078 {
4410079 //
4410080 // The environment item was found; if it can
4410081 // be overwritten,
4410082 // the write is done.
4410083 //
4410084 if (overwrite)
4410085 {
4410086 //
4410087 // To be able to handle both 'setenv()'
4410088 // and 'putenv()',
4410089 // before removing the item, it is fixed
4410090 // the pointer to
4410091 // the global environment table.
4410092 //
4410093 _environment[e] = _environment_table[e];
4410094 //
4410095 // Now copy the new environment. The
4410096 // string size was
4410097 // already checked.
4410098 //
4410099 strcpy (_environment[e], name);
4410100 strcat (_environment[e], "=");
4410101 strcat (_environment[e], value);
4410102 //
4410103 // Return.
4410104 //
4410105 return (0);
4410106 }
4410107 //
4410108 // Cannot overwrite!
```

```
4410109         //
4410110         errset (EUNKNOWN);
4410111         return (-1);
4410112     }
4410113 }
4410114 //
4410115 // The item was not found. Scan again for a free
4410116 // slot.
4410117 //
4410118 for (e = 0; e < ARG_MAX / 32; e++)
4410119 {
4410120     if (_environment[e] == NULL
4410121         || _environment[e][0] == 0)
4410122     {
4410123         //
4410124         // An empty item was found. To be able to
4410125         // handle both
4410126         // 'setenv()' and 'putenv()', it is fixed
4410127         // the pointer to
4410128         // the global environment table.
4410129         //
4410130         _environment[e] = _environment_table[e];
4410131         //
4410132         // Now copy the new environment. The string
4410133         // size was
4410134         // already checked.
4410135         //
4410136         strcpy (_environment[e], name);
4410137         strcat (_environment[e], "=");
4410138         strcat (_environment[e], value);
4410139         //
4410140         // Return.
4410141         //
4410142         return (0);
4410143     }
4410144 }
4410145 //
```

```
4410146 // Sorry: the empty slot was not found!
4410147 //
4410148 errset (ENOMEM); // Not enough space.
4410149 return (-1);
4410150 }
```

95.19.19 lib/stdlib/strtol.c



Si veda la sezione [88.130](#).

```
4420001 #include <stdlib.h>
4420002 #include <ctype.h>
4420003 #include <errno.h>
4420004 #include <limits.h>
4420005 #include <stdbool.h>
4420006 //-----
4420007 #define isoctal(C) ((int) (C >= '0' && C <= '7'))
4420008 //-----
4420009 long int
4420010 strtol (const char *restrict string,
4420011         char **restrict endptr, int base)
4420012 {
4420013     int i;
4420014     int sign = +1;
4420015     long int number;
4420016     long int previous;
4420017     int digit;
4420018     //
4420019     bool flag_prefix_oct = 0;
4420020     bool flag_prefix_exa = 0;
4420021     bool flag_prefix_dec = 0;
4420022     //
4420023     // Check base and string.
4420024     //
4420025     // With base 1 cannot do anything.
4420026     //
4420027     if (base < 0 || base > 36 || base == 1
```

```
4420028     || string == NULL || string[0] == 0)
4420029     {
4420030         if (endptr != NULL)
4420031             *endptr = (char *) string;
4420032         errset (EINVAL); // Invalid argument.
4420033         return ((long int) 0);
4420034     }
4420035     //
4420036     // Eat initial spaces.
4420037     //
4420038     for (i = 0; isspace (string[i]); i++)
4420039     {
4420040         ;
4420041     }
4420042     //
4420043     // Check sign.
4420044     //
4420045     if (string[i] == '+')
4420046     {
4420047         sign = +1;
4420048         i++;
4420049     }
4420050     else if (string[i] == '-')
4420051     {
4420052         sign = -1;
4420053         i++;
4420054     }
4420055     //
4420056     // Check for prefix.
4420057     //
4420058     if (string[i] == '0')
4420059     {
4420060         if (string[i + 1] == 'x' || string[i + 1] == 'X')
4420061         {
4420062             flag_prefix_exa = 1;
4420063         }
4420064         else if (isoctal (string[i + 1]))
```

```
4420065     {
4420066         flag_prefix_oct = 1;
4420067     }
4420068     else
4420069     {
4420070         flag_prefix_dec = 1;
4420071     }
4420072 }
4420073 else if (isdigit (string[i]))
4420074 {
4420075     flag_prefix_dec = 1;
4420076 }
4420077 //
4420078 // Check compatibility with requested base.
4420079 //
4420080 if (flag_prefix_exa)
4420081 {
4420082     //
4420083     // At the moment, there is a zero and a 'x'.
4420084     // Might be
4420085     // exadecimal, or might be a number base 33 or
4420086     // more.
4420087     //
4420088     if (base == 0)
4420089     {
4420090         base = 16;
4420091     }
4420092     else if (base == 16)
4420093     {
4420094         ;    // Ok.
4420095     }
4420096     else if (base >= 33)
4420097     {
4420098         ;    // Ok.
4420099     }
4420100     else
4420101     {
```

```
4420102         //
4420103         // Incompatible sequence: only the initial
4420104         // zero is reported.
4420105         //
4420106         if (endptr != NULL)
4420107             *endptr = (char *) &string[i + 1];
4420108         return ((long int) 0);
4420109     }
4420110     //
4420111     // Move on, after the '0x' prefix.
4420112     //
4420113     i += 2;
4420114 }
4420115 //
4420116 if (flag_prefix_oct)
4420117 {
4420118     //
4420119     // There is a zero and a digit.
4420120     //
4420121     if (base == 0)
4420122     {
4420123         base = 8;
4420124     }
4420125     //
4420126     // Move on, after the '0' prefix.
4420127     //
4420128     i += 1;
4420129 }
4420130 //
4420131 if (flag_prefix_dec)
4420132 {
4420133     if (base == 0)
4420134     {
4420135         base = 10;
4420136     }
4420137 }
4420138 //
```

```
4420139 // Scan the string.
4420140 //
4420141 for (number = 0; string[i] != 0; i++)
4420142 {
4420143     if (string[i] >= '0' && string[i] <= '9')
4420144     {
4420145         digit = string[i] - '0';
4420146     }
4420147     else if (string[i] >= 'A' && string[i] <= 'Z')
4420148     {
4420149         digit = string[i] - 'A' + 10;
4420150     }
4420151     else if (string[i] >= 'a' && string[i] <= 'z')
4420152     {
4420153         digit = string[i] - 'a' + 10;
4420154     }
4420155     else
4420156     {
4420157         //
4420158         // This is an out of range digit.
4420159         //
4420160         digit = 999;
4420161     }
4420162     //
4420163     // Give a sign to the digit.
4420164     //
4420165     digit *= sign;
4420166     //
4420167     // Compare with the base.
4420168     //
4420169     if (base > (digit * sign))
4420170     {
4420171         //
4420172         // Check if the current digit can be safely
4420173         // computed.
4420174         //
4420175         previous = number;
```

```
4420176     number *= base;
4420177     number += digit;
4420178     if (number / base != previous)
4420179     {
4420180         //
4420181         // Out of range.
4420182         //
4420183         if (endptr != NULL)
4420184             *endptr = (char *) &string[i + 1];
4420185         errset (ERANGE); // Result too large.
4420186         if (sign > 0)
4420187             {
4420188                 return (LONG_MAX);
4420189             }
4420190         else
4420191             {
4420192                 return (LONG_MIN);
4420193             }
4420194     }
4420195 }
4420196 else
4420197 {
4420198     if (endptr != NULL)
4420199         *endptr = (char *) &string[i];
4420200     return (number);
4420201 }
4420202 }
4420203 //
4420204 // The string is finished.
4420205 //
4420206 if (endptr != NULL)
4420207     *endptr = (char *) &string[i];
4420208 //
4420209 return (number);
4420210 }
```


95.19.20 lib/stdlib/strtoul.c



Si veda la sezione [88.130](#).

```
4430001 #include <stdlib.h>
4430002 #include <ctype.h>
4430003 #include <errno.h>
4430004 #include <limits.h>
4430005 //-----
4430006 // A really poor implementation. ,-(
4430007 //
4430008 unsigned long int
4430009 strtoul (const char *restrict string,
4430010         char **restrict endptr, int base)
4430011 {
4430012     return ((unsigned long int)
4430013           strtol (string, endptr, base));
4430014 }
```

95.19.21 lib/stdlib/unsetenv.c



Si veda la sezione [88.104](#).

```
4440001 #include <stdlib.h>
4440002 #include <string.h>
4440003 #include <errno.h>
4440004 //-----
4440005 extern char *_environment[];
4440006 extern char *_environment_table[];
4440007 //-----
4440008 int
4440009 unsetenv (const char *name)
4440010 {
4440011     int e;           // First index: environment table
4440012     // items.
4440013     int f;           // Second index: environment string
4440014     // scan.
4440015     //
```

```
4440016 // Check if the input is empty. No error is
4440017 // reported.
4440018 //
4440019 if (name == NULL || strlen (name) == 0)
4440020 {
4440021     return (0);
4440022 }
4440023 //
4440024 // Check if the input is valid: error here is
4440025 // reported.
4440026 //
4440027 if (strchr (name, '=') != NULL)
4440028 {
4440029     errset (EINVAL); // Invalid argument.
4440030     return (-1);
4440031 }
4440032 //
4440033 // Scan the environment table items, with index 'e'.
4440034 //
4440035 for (e = 0; e < ARG_MAX / 32; e++)
4440036 {
4440037     //
4440038     // Scan the string of the environment item, with
4440039     // index 'f'.
4440040     // The scan continue until 'name[f]' and
4440041     // '_environment[e][f]'
4440042     // are equal.
4440043     //
4440044     for (f = 0;
4440045          f < ARG_MAX / 16 - 1
4440046          && name[f] == _environment[e][f]; f++)
4440047     {
4440048         ; // Just scan.
4440049     }
4440050     //
4440051     // At this point, 'name[f]' and
4440052     // '_environment[e][f]' are
```

```
4440053 // different: if 'name[f]' is zero the name
4440054 // string is
4440055 // terminated; if '_environment[e][f]' is also
4440056 // equal to '=',
4440057 // the environment item is corresponding to the
4440058 // requested name.
4440059 //
4440060 if (name[f] == 0 && _environment[e][f] == '=')
4440061 {
4440062     //
4440063     // The environment item was found and it
4440064     // have to be removed.
4440065     // To be able to handle both 'setenv()' and
4440066     // 'putenv()',
4440067     // before removing the item, it is fixed the
4440068     // pointer to
4440069     // the global environment table.
4440070     //
4440071     _environment[e] = _environment_table[e];
4440072     //
4440073     // Now remove the environment item.
4440074     //
4440075     _environment[e][0] = 0;
4440076     break;
4440077 }
4440078 }
4440079 //
4440080 // Work done fine.
4440081 //
4440082 return (0);
4440083 }
```

95.19.22 lib/stdlib_alloc/_alloc_list.c



Si veda la sezione [88.76](#).

```
4450001 #include <stdlib.h>
4450002 #include <stdio.h>
4450003 #include <unistd.h>
4450004 #include <stdint.h>
4450005 //-----
4450006 extern uintptr_t _alloc_start;
4450007 //-----
4450008 void
4450009 _alloc_list (void)
4450010 {
4450011     uintptr_t start = _alloc_start;
4450012     uintptr_t end = (uintptr_t) sbrk (0);
4450013     _alloc_head_t *head = (void *) start;
4450014     size_t actual_size;
4450015     uintptr_t current;
4450016     uintptr_t next;
4450017     uintptr_t up_to;
4450018     int counter;
4450019     //
4450020     // Scandisce la lista di blocchi di memoria.
4450021     //
4450022     counter = 2;
4450023     while (counter)
4450024     {
4450025         //
4450026         // Annota la posizione attuale e quella
4450027         // successiva.
4450028         //
4450029         current = (uintptr_t) head;
4450030         next = head->next * (sizeof (_alloc_head_t));
4450031         if (next == start)
4450032         {
4450033             up_to = end;
4450034         }
```

```
4450035     else
4450036     {
4450037         up_to = next;
4450038     }
4450039     //
4450040     // Se è stato raggiunto il primo elemento,
4450041     // decrementa il
4450042     // contatore di una unità. Se è già a zero,
4450043     // esce.
4450044     //
4450045     if (current == start)
4450046     {
4450047         counter--;
4450048         if (counter == 0)
4450049             break;
4450050     }
4450051     //
4450052     // Determina la dimensione del blocco attuale.
4450053     //
4450054     if (current == start && next == start)
4450055     {
4450056         //
4450057         // Si tratta del primo e unico elemento
4450058         // della lista.
4450059         //
4450060         actual_size =
4450061             end - start - (sizeof (_alloc_head_t));
4450062     }
4450063     else
4450064     {
4450065         actual_size =
4450066             up_to - current - (sizeof (_alloc_head_t));
4450067     }
4450068     //
4450069     // Si mostra lo stato del blocco di memoria.
4450070     //
4450071     if (head->allocated)
```

```
4450072     {
4450073         printf ("%s] used %08X..%08X size %08zX\n",
4450074             __func__,
4450075             current + (sizeof (_alloc_head_t)),
4450076             up_to, actual_size);
4450077     }
4450078     else
4450079     {
4450080         printf ("%s] free %08X..%08X size %08zX\n",
4450081             __func__,
4450082             current + (sizeof (_alloc_head_t)),
4450083             up_to, actual_size);
4450084     }
4450085     //
4450086     // Si passa alla posizione successiva.
4450087     //
4450088     head = (void *) next;
4450089 }
4450090 }
```

95.19.23 lib/stdlib_alloc/free.c

<<

Si veda la sezione [88.76](#).

```
4460001 #include <stdlib.h>
4460002 #include <stdio.h>
4460003 #include <unistd.h>
4460004 //-----
4460005 extern uintptr_t _alloc_start;
4460006 //-----
4460007 void
4460008 free (void *ptr)
4460009 {
4460010     _alloc_head_t *start = (_alloc_head_t *) _alloc_start;
4460011     _alloc_head_t *head_current = ((_alloc_head_t *) ptr) - 1;
4460012     _alloc_head_t *head_next;
4460013     //
```

```
4460014 // Verifica il blocco attuale e, se è possibile, lo
4460015 // libera.
4460016 //
4460017 if (head_current->allocated == 1)
4460018 {
4460019     head_current->allocated = 0;
4460020 }
4460021 else
4460022 {
4460023     printf ("%s] ERROR: cannot free %08X!\n",
4460024             __func__,
4460025             (uintptr_t) head_current +
4460026             (sizeof (_alloc_head_t)));
4460027 }
4460028 //
4460029 // Scandisce i blocchi liberi, cercando quelli
4460030 // adiacenti per
4460031 // allungarli. Se il blocco successivo è il primo,
4460032 // termina,
4460033 // perché non può avvenire alcuna fusione con
4460034 // quello precedente.
4460035 //
4460036 head_current = start;
4460037 while (1)
4460038 {
4460039     //
4460040     // Individua il blocco successivo.
4460041     //
4460042     head_next =
4460043         (_alloc_head_t *) (head_current->next
4460044                             * (sizeof (_alloc_head_t)));
4460045     //
4460046     // Controlla se è il primo.
4460047     //
4460048     if (head_next == start)
4460049     {
4460050         break;
```

```
4460051     }
4460052     //
4460053     //
4460054     //
4460055     if (head_current->allocated == 0)
4460056     {
4460057         //
4460058         // Controlla se si può espandere.
4460059         //
4460060         if (head_next->allocated == 0)
4460061         {
4460062             head_current->next = head_next->next;
4460063         }
4460064         else
4460065         {
4460066             head_current = head_next;
4460067         }
4460068     }
4460069     else
4460070     {
4460071         head_current = (_alloc_head_t *)
4460072             (head_current->next * (sizeof (_alloc_head_t)));
4460073     }
4460074 }
4460075 }
```

95.19.24 lib/stdlib_alloc/malloc.c



Si veda la sezione [88.76](#).

```
4470001 #include <stdlib.h>
4470002 #include <unistd.h>
4470003 #include <errno.h>
4470004 //-----
4470005 uintptr_t _alloc_start = 0;
4470006 //-----
4470007 static int _alloc_init (void);
```



```
4470008 static void *_malloc (size_t size);
4470009 //-----
4470010 void *
4470011 malloc (size_t size)
4470012 {
4470013     void *pstatus;
4470014     int status;
4470015     //
4470016     // Verify to have initialized the allocation memory.
4470017     //
4470018     if (_alloc_start == 0)
4470019     {
4470020         status = _alloc_init ();
4470021         if (status < 0)
4470022         {
4470023             errset (ENOMEM);
4470024             return (NULL);
4470025         }
4470026     }
4470027     //
4470028     // Try to allocate as usual.
4470029     //
4470030     pstatus = _malloc (size);
4470031     //
4470032     if (pstatus == NULL)
4470033     {
4470034         //
4470035         // Try to increase memory for the process.
4470036         //
4470037         pstatus = sbrk (size);
4470038         if (pstatus == NULL)
4470039         {
4470040             //
4470041             // Sorry: no way to get memory.
4470042             //
4470043             errset (ENOMEM);
4470044             return (NULL);
```

```
4470045     }
4470046     //
4470047     // Ok. Now try again to allocate memory.
4470048     //
4470049     return (_malloc (size));
4470050 }
4470051 else
4470052 {
4470053     //
4470054     // The first allocation was successful.
4470055     //
4470056     return (pstatus);
4470057 }
4470058 }
4470059
4470060 //-----
4470061 static int
4470062 _alloc_init (void)
4470063 {
4470064     uintptr_t start;
4470065     uintptr_t end;
4470066     _alloc_head_t *head;
4470067     size_t available;
4470068     //
4470069     // Get size.
4470070     //
4470071     if (_alloc_start == 0)
4470072     {
4470073         _alloc_start = (uintptr_t) sbrk (0);
4470074     }
4470075     //
4470076     start = _alloc_start;
4470077     end = (uintptr_t) sbrk (0);
4470078     available = end - start;
4470079     //
4470080     // Check available space.
4470081     //
```

```
4470082     if (available < ((sizeof (_alloc_head_t)) * 2))
4470083     {
4470084         //
4470085         // Try to get a little memory.
4470086         //
4470087         sbrk ((sizeof (_alloc_head_t)) * 2);
4470088         end = (uintptr_t) sbrk (0);
4470089         available = end - start;
4470090         if (available < ((sizeof (_alloc_head_t)) * 2))
4470091         {
4470092             //
4470093             // Sorry!
4470094             //
4470095             return (-1);
4470096         }
4470097     }
4470098     //
4470099     // Prepare the list main node.
4470100     //
4470101     head = (_alloc_head_t *) start;
4470102     //
4470103     // Init the first free block, that points to itself,
4470104     // as it is
4470105     // the only one.
4470106     //
4470107     head->allocated = 0;
4470108     head->next = (start / (sizeof (_alloc_head_t)));
4470109     //
4470110     // Ok.
4470111     //
4470112     return (0);
4470113 }
4470114
4470115 //-----
4470116 static void *
4470117 _malloc (size_t size)
4470118 {
```

```
4470119  uintptr_t start = _alloc_start;
4470120  uintptr_t end = (uintptr_t) sbrk (0);
4470121  _alloc_head_t *head = (void *) start;
4470122  size_t actual_size;
4470123  uintptr_t current;
4470124  uintptr_t next;
4470125  uintptr_t new;
4470126  uintptr_t up_to;
4470127  int counter;
4470128  //
4470129  // Arrotonda in eccesso il valore di «size», in
4470130  // modo che sia un
4470131  // multiplo della dimensione di «_alloc_head_t».
4470132  // Altrimenti, la
4470133  // collocazione dei blocchi successivi può avvenire
4470134  // in modo
4470135  // non allineato.
4470136  //
4470137  size = (size + (sizeof (_alloc_head_t)) - 1);
4470138  size = size / (sizeof (_alloc_head_t));
4470139  size = size * (sizeof (_alloc_head_t));
4470140  //
4470141  // Cerca un blocco libero di dimensione sufficiente.
4470142  //
4470143  counter = 2;
4470144  while (counter)
4470145  {
4470146      //
4470147      // Annota la posizione attuale e quella
4470148      // successiva.
4470149      //
4470150      current = (uintptr_t) head;
4470151      next = head->next * (sizeof (_alloc_head_t));
4470152      //
4470153      if (next == start)
4470154      {
4470155          up_to = end;
```

```
4470156     }
4470157     else
4470158     {
4470159         up_to = next;
4470160     }
4470161     //
4470162     // Se è stato raggiunto il primo elemento,
4470163     // decrementa il
4470164     // contatore di una unità. Se è già a zero,
4470165     // esce.
4470166     //
4470167     if (current == start)
4470168     {
4470169         counter--;
4470170         if (counter == 0)
4470171             break;
4470172     }
4470173     //
4470174     // Controlla se si tratta di un blocco libero.
4470175     //
4470176
4470177     if (!head->allocated)
4470178     {
4470179         //
4470180         // Il blocco è libero: si deve determinarne
4470181         // la dimensione.
4470182         //
4470183         if (current == start && next == start)
4470184         {
4470185             //
4470186             // Si tratta del primo e unico elemento
4470187             // della lista.
4470188             //
4470189             actual_size =
4470190                 end - start - (sizeof (_alloc_head_t));
4470191         }
4470192     else
```

```
4470193     {
4470194         actual_size =
4470195             up_to - current - (sizeof (_alloc_head_t));
4470196     }
4470197     //
4470198     // Si verifica che sia capiente.
4470199     //
4470200     if (actual_size >=
4470201         size + ((sizeof (_alloc_head_t)) * 2))
4470202     {
4470203         //
4470204         // C'è spazio per dividere il blocco.
4470205         //
4470206         new =
4470207             current + size + (sizeof (_alloc_head_t));
4470208         //
4470209         // Aggiorna l'intestazione attuale.
4470210         //
4470211         head->allocated = 1;
4470212         head->next = new / (sizeof (_alloc_head_t));
4470213         //
4470214         // Predisporre l'intestazione successiva.
4470215         //
4470216         head = (void *) new;
4470217         head->allocated = 0;
4470218         head->next = next / (sizeof (_alloc_head_t));
4470219         //
4470220         // Restituisce l'indirizzo iniziale
4470221         // dello spazio libero,
4470222         // successivo all'intestazione.
4470223         //
4470224         return (void *) (current +
4470225                         (sizeof (_alloc_head_t)));
4470226     }
4470227     else if (actual_size >= size)
4470228     {
4470229         //
```

```
4470230 // Il blocco va usato per intero.
4470231 //
4470232 head->allocated = 1;
4470233 //
4470234 // Restituisce l'indirizzo iniziale
4470235 // dello spazio libero,
4470236 // successivo all'intestazione.
4470237 //
4470238 return (void *) (current +
4470239                 (sizeof (_alloc_head_t)));
4470240     }
4470241 }
4470242 //
4470243 // Il blocco è allocato, oppure è di
4470244 // dimensione insufficiente;
4470245 // pertanto occorre passare alla posizione
4470246 // successiva.
4470247 //
4470248 head = (void *) next;
4470249 }
4470250 //
4470251 // Essendo terminato il ciclo precedente, vuol dire
4470252 // che non ci sono spazi disponibili.
4470253 //
4470254 errset (ENOMEM);
4470255 return NULL;
4470256 }
```

95.19.25 lib/stdlib_alloc/realloc.c

Si veda la sezione [88.76](#).

```
4480001 #include <stdlib.h>
4480002 #include <stdio.h>
4480003 #include <unistd.h>
4480004 #include <string.h>
4480005 //-----
```

```
4480006 extern uintptr_t _alloc_start;
4480007 //-----
4480008 void *
4480009 realloc (void *ptr, size_t size)
4480010 {
4480011     uintptr_t start = _alloc_start;
4480012     uintptr_t end = (uintptr_t) sbrk (0);
4480013     size_t actual_size;
4480014     _alloc_head_t *head = ((_alloc_head_t *) ptr) - 1;
4480015     _alloc_head_t *head_new;
4480016     void *ptr_new;
4480017     //
4480018     // Verifica che il puntatore riguardi effettivamente
4480019     // un'area occupata.
4480020     //
4480021     if (!head->allocated)
4480022     {
4480023         printf
4480024             ("%s] ERROR: cannot re-allocate %08X that is "
4480025              "not already allocated!", __func__,
4480026              (uintptr_t) ptr);
4480027     }
4480028     //
4480029     // Arrotonda in eccesso il valore di «size», in
4480030     // modo che sia un
4480031     // multiplo della dimensione di «_alloc_head_t».
4480032     // Altrimenti, la
4480033     // collocazione dei blocchi successivi può avvenire
4480034     // in modo
4480035     // non allineato.
4480036     //
4480037     size = (size + (sizeof (_alloc_head_t)) - 1);
4480038     size = size / (sizeof (_alloc_head_t));
4480039     size = size * (sizeof (_alloc_head_t));
4480040     //
4480041     // Determina la dimensione attuale.
4480042     //
```



```
4480043     if ((head->next * (sizeof (_alloc_head_t))) == start)
4480044     {
4480045         actual_size = end - ((uintptr_t) ptr);
4480046     }
4480047     else
4480048     {
4480049         actual_size =
4480050             (head->next * (sizeof (_alloc_head_t))) -
4480051             ((uintptr_t) ptr);
4480052     }
4480053     //
4480054     // Se la dimensione richiesta è inferiore, può
4480055     // ridurre
4480056     // l'estensione del blocco.
4480057     //
4480058     if (size == actual_size)
4480059     {
4480060         return ptr;
4480061     }
4480062     else if (size <=
4480063             (actual_size - (sizeof (_alloc_head_t)) * 2))
4480064     {
4480065         //
4480066         // Si può ricavare lo spazio libero rimanente.
4480067         //
4480068         head_new = (_alloc_head_t *) (((char *) ptr) + size);
4480069         //
4480070         head_new->next = head->next;
4480071         head_new->allocated = 0;
4480072         //
4480073         head->next =
4480074             ((uintptr_t) head_new) / (sizeof (_alloc_head_t));
4480075         //
4480076         return ptr;
4480077     }
4480078     else if (size < actual_size)
4480079     {
```

```
4480080 //
4480081 // Anche se è minore, non si può ridurre lo
4480082 // spazio usato
4480083 // effettivamente.
4480084 //
4480085 return ptr;
4480086 }
4480087 else
4480088 {
4480089 //
4480090 // La dimensione richiesta è maggiore.
4480091 //
4480092 ptr_new = malloc (size);
4480093 //
4480094 if (ptr_new)
4480095 {
4480096 //
4480097 // Ricopia i dati nella nuova collocazione.
4480098 //
4480099 memcpy (ptr_new, ptr, actual_size);
4480100 //
4480101 // Libera la collocazione vecchia.
4480102 //
4480103 free (ptr);
4480104 //
4480105 return ptr_new;
4480106 }
4480107 else
4480108 {
4480109 return NULL;
4480110 }
4480111 }
4480112 }
```

95.20 os32: «lib/string.h»



Si veda la sezione [91.3](#).

```
4490001 #ifndef _STRING_H
4490002 #define _STRING_H      1
4490003 //-----
4490004 #include <size_t.h>
4490005 #include <NULL.h>
4490006 #include <restrict.h>
4490007 //-----
4490008 void *memccpy (void *restrict dst,
4490009               const void *restrict org, int c, size_t n);
4490010 void *memchr (const void *memory, int c, size_t n);
4490011 int memcmp (const void *memory1, const void *memory2,
4490012            size_t n);
4490013 void *memcpy (void *restrict dst,
4490014              const void *restrict org, size_t n);
4490015 void *memmove (void *dst, const void *org, size_t n);
4490016 void *memset (void *memory, int c, size_t n);
4490017 char *strcat (char *restrict dst, const char *restrict org);
4490018 char *strchr (const char *string, int c);
4490019 int strcmp (const char *string1, const char *string2);
4490020 int strcoll (const char *string1, const char *string2);
4490021 char *strcpy (char *restrict dst, const char *restrict org);
4490022 size_t strcspn (const char *string, const char *reject);
4490023 char *strdup (const char *string);
4490024 char *strerror (int errnum);
4490025 size_t strlen (const char *string);
4490026 char *strncat (char *restrict dst,
4490027               const char *restrict org, size_t n);
4490028 int strncmp (const char *string1, const char *string2,
4490029             size_t n);
4490030 char *strncpy (char *restrict dst,
4490031               const char *restrict org, size_t n);
4490032 char *strpbrk (const char *string, const char *accept);
4490033 char *strrchr (const char *string, int c);
4490034 size_t strspn (const char *string, const char *accept);
```

```

4490035 char *strstr (const char *string, const char *substring);
4490036 char *strtok (char *restrict string,
4490037             const char *restrict delim);
4490038 size_t strxfrm (char *restrict dst,
4490039              const char *restrict org, size_t n);
4490040 //-----
4490041
4490042 #endif

```

95.20.1	lib/string/memccpy.c	2069
95.20.2	lib/string/memchr.c	2070
95.20.3	lib/string/memcmp.c	2070
95.20.4	lib/string/memcpy.c	2071
95.20.5	lib/string/memmove.c	2071
95.20.6	lib/string/memset.c	2073
95.20.7	lib/string/streac.c	2073
95.20.8	lib/string/strchr.c	2074
95.20.9	lib/string/strcmp.c	2074
95.20.10	lib/string/strcoll.c	2075
95.20.11	lib/string/strcpy.c	2075
95.20.12	lib/string/strcspn.c	2076
95.20.13	lib/string/strdup.c	2077
95.20.14	lib/string/strerror.c	2077
95.20.15	lib/string/strlen.c	2081
95.20.16	lib/string/strncat.c	2082

95.20.17	lib/string/strncmp.c	2082
95.20.18	lib/string/strncpy.c	2083
95.20.19	lib/string/strpbrk.c	2084
95.20.20	lib/string/strrchr.c	2084
95.20.21	lib/string/strspn.c	2085
95.20.22	lib/string/strstr.c	2086
95.20.23	lib/string/strtok.c	2087
95.20.24	lib/string/strxfrm.c	2091

95.20.1 lib/string/memccpy.c



Si veda la sezione [88.77](#).

```
4500001 #include <string.h>
4500002 //-----
4500003 void *
4500004 memccpy (void *restrict dst, const void *restrict org,
4500005          int c, size_t n)
4500006 {
4500007     char *d = (char *) dst;
4500008     char *o = (char *) org;
4500009     size_t i;
4500010     for (i = 0; n > 0 && i < n; i++)
4500011     {
4500012         d[i] = o[i];
4500013         if (d[i] == (char) c)
4500014         {
4500015             return ((void *) &d[i + 1]);
4500016         }
4500017     }
4500018     return (NULL);
4500019 }
```

95.20.2 lib/string/memchr.c



Si veda la sezione [88.78](#).

```
4510001 #include <string.h>
4510002 //-----
4510003 void *
4510004 memchr (const void *memory, int c, size_t n)
4510005 {
4510006     char *m = (char *) memory;
4510007     size_t i;
4510008     for (i = 0; n > 0 && i < n; i++)
4510009     {
4510010         if (m[i] == (char) c)
4510011         {
4510012             return (void *) (m + i);
4510013         }
4510014     }
4510015     return NULL;
4510016 }
```

95.20.3 lib/string/memcmp.c



Si veda la sezione [88.79](#).

```
4520001 #include <string.h>
4520002 //-----
4520003 int
4520004 memcmp (const void *memory1, const void *memory2, size_t n)
4520005 {
4520006     char *a = (char *) memory1;
4520007     char *b = (char *) memory2;
4520008     size_t i;
4520009     for (i = 0; n > 0 && i < n; i++)
4520010     {
4520011         if (a[i] > b[i])
4520012         {
4520013             return 1;
4520014         }
4520015     }
4520016     return 0;
4520017 }
```

```
4520014     }
4520015     else if (a[i] < b[i])
4520016     {
4520017         return -1;
4520018     }
4520019 }
4520020 return 0;
4520021 }
```

95.20.4 lib/string/memcpy.c



Si veda la sezione [88.80](#).

```
4530001 #include <string.h>
4530002 //-----
4530003 void *
4530004 memcpy (void *restrict dst, const void *restrict org,
4530005         size_t n)
4530006 {
4530007     char *d = (char *) dst;
4530008     char *o = (char *) org;
4530009     size_t i;
4530010     for (i = 0; n > 0 && i < n; i++)
4530011     {
4530012         d[i] = o[i];
4530013     }
4530014     return dst;
4530015 }
```

95.20.5 lib/string/memmove.c



Si veda la sezione [88.81](#).

```
4540001 #include <string.h>
4540002 //-----
4540003 void *
4540004 memmove (void *dst, const void *org, size_t n)
```

```
4540005 {
4540006     char *d = (char *) dst;
4540007     char *o = (char *) org;
4540008     size_t i;
4540009     //
4540010     // Depending on the memory start locations, copy may
4540011     // be direct or
4540012     // reverse, to avoid overwriting before the
4540013     // relocation is done.
4540014     //
4540015     if (d < o)
4540016     {
4540017         for (i = 0; i < n; i++)
4540018         {
4540019             d[i] = o[i];
4540020         }
4540021     }
4540022     else if (d == o)
4540023     {
4540024         //
4540025         // Memory locations are already the same.
4540026         //
4540027         ;
4540028     }
4540029     else
4540030     {
4540031         for (i = n - 1; i >= 0; i--)
4540032         {
4540033             d[i] = o[i];
4540034         }
4540035     }
4540036     return dst;
4540037 }
```


95.20.6 lib/string/memset.c



Si veda la sezione [88.82](#).

```
4550001 #include <string.h>
4550002 //-----
4550003 void *
4550004 memset (void *memory, int c, size_t n)
4550005 {
4550006     char *m = (char *) memory;
4550007     size_t i;
4550008     for (i = 0; n > 0 && i < n; i++)
4550009     {
4550010         m[i] = (char) c;
4550011     }
4550012     return memory;
4550013 }
```

95.20.7 lib/string/strcat.c



Si veda la sezione [88.113](#).

```
4560001 #include <string.h>
4560002 //-----
4560003 char *
4560004 strcat (char *restrict dst, const char *restrict org)
4560005 {
4560006     size_t i;
4560007     size_t j;
4560008     for (i = 0; dst[i] != 0; i++)
4560009     {
4560010         ; // Just look for the null character.
4560011     }
4560012     for (j = 0; org[j] != 0; i++, j++)
4560013     {
4560014         dst[i] = org[j];
4560015     }
4560016     dst[i] = 0;
```

```
4560017     return dst;
4560018 }
```

95.20.8 lib/string/strchr.c



Si veda la sezione [88.114](#).

```
4570001 #include <string.h>
4570002 //-----
4570003 char *
4570004 strchr (const char *string, int c)
4570005 {
4570006     size_t i;
4570007     for (i = 0;; i++)
4570008     {
4570009         if (string[i] == (char) c)
4570010         {
4570011             return (char *) (string + i);
4570012         }
4570013         else if (string[i] == 0)
4570014         {
4570015             return NULL;
4570016         }
4570017     }
4570018 }
```

95.20.9 lib/string/strcmp.c



Si veda la sezione [88.115](#).

```
4580001 #include <string.h>
4580002 //-----
4580003 int
4580004 strcmp (const char *string1, const char *string2)
4580005 {
4580006     char *a = (char *) string1;
4580007     char *b = (char *) string2;
```

```
4580008     size_t i;
4580009     for (i = 0;; i++)
4580010     {
4580011         if (a[i] > b[i])
4580012         {
4580013             return 1;
4580014         }
4580015         else if (a[i] < b[i])
4580016         {
4580017             return -1;
4580018         }
4580019         else if (a[i] == 0 && b[i] == 0)
4580020         {
4580021             return 0;
4580022         }
4580023     }
4580024 }
```

95.20.10 lib/string/strcoll.c

Si veda la sezione [88.115](#).

```
4590001     #include <string.h>
4590002     //-----
4590003     int
4590004     strcoll (const char *string1, const char *string2)
4590005     {
4590006         return (strcmp (string1, string2));
4590007     }
```

95.20.11 lib/string/strcpy.c

Si veda la sezione [88.117](#).

```
4600001     #include <string.h>
4600002     //-----
4600003     char *
```

```
4600004 strcpy (char *restrict dst, const char *restrict org)
4600005 {
4600006     size_t i;
4600007     for (i = 0; org[i] != 0; i++)
4600008     {
4600009         dst[i] = org[i];
4600010     }
4600011     dst[i] = 0;
4600012     return dst;
4600013 }
```

95.20.12 lib/string/strcspn.c



Si veda la sezione [88.127](#).

```
4610001 #include <string.h>
4610002 //-----
4610003 size_t
4610004 strcspn (const char *string, const char *reject)
4610005 {
4610006     size_t i;
4610007     size_t j;
4610008     int found;
4610009     for (i = 0; string[i] != 0; i++)
4610010     {
4610011         for (j = 0, found = 0; reject[j] != 0 || found; j++)
4610012         {
4610013             if (string[i] == reject[j])
4610014             {
4610015                 found = 1;
4610016                 break;
4610017             }
4610018         }
4610019         if (found)
4610020         {
4610021             break;
4610022         }
4610023     }
4610024 }
```

```
4610023     }
4610024     return i;
4610025 }
```

95.20.13 lib/string/strdup.c

Si veda la sezione [88.119](#).

```
4620001 #include <string.h>
4620002 #include <stdlib.h>
4620003 #include <errno.h>
4620004 //-----
4620005 char *
4620006 strdup (const char *string)
4620007 {
4620008     size_t size;
4620009     char *copy;
4620010     //
4620011     // Get string size: must be added 1, to count the
4620012     // termination null
4620013     // character.
4620014     //
4620015     size = strlen (string) + 1;
4620016     //
4620017     copy = malloc (size);
4620018     //
4620019     if (copy == NULL)
4620020     {
4620021         errset (ENOMEM); // Not enough memory.
4620022         return (NULL);
4620023     }
4620024     //
4620025     strcpy (copy, string);
4620026     //
4620027     return (copy);
4620028 }
```

95.20.14 lib/string/strerror.c



Si veda la sezione [88.120](#).

```
4630001 #include <string.h>
4630002 #include <errno.h>
4630003 //-----
4630004 #define ERROR_MAX 120
4630005 //-----
4630006 char *
4630007 strerror (int errnum)
4630008 {
4630009     static char *err[ERROR_MAX];
4630010     //
4630011     err[0] = "No error";
4630012     err[E2BIG] = TEXT_E2BIG;
4630013     err[EACCES] = TEXT_EACCES;
4630014     err[EADDRINUSE] = TEXT_EADDRINUSE;
4630015     err[EADDRNOTAVAIL] = TEXT_EADDRNOTAVAIL;
4630016     err[EAFNOSUPPORT] = TEXT_EAFNOSUPPORT;
4630017     err[EAGAIN] = TEXT_EAGAIN;
4630018     err[EALREADY] = TEXT_EALREADY;
4630019     err[EBADF] = TEXT_EBADF;
4630020     err[EBADMSG] = TEXT_EBADMSG;
4630021     err[EBUSY] = TEXT_EBUSY;
4630022     err[ECANCELED] = TEXT_ECANCELED;
4630023     err[ECHILD] = TEXT_ECHILD;
4630024     err[ECONNABORTED] = TEXT_ECONNABORTED;
4630025     err[ECONNREFUSED] = TEXT_ECONNREFUSED;
4630026     err[ECONNRESET] = TEXT_ECONNRESET;
4630027     err[EDEADLK] = TEXT_EDEADLK;
4630028     err[EDESTADDRREQ] = TEXT_EDESTADDRREQ;
4630029     err[EDOM] = TEXT_EDOM;
4630030     err[EDQUOT] = TEXT_EDQUOT;
4630031     err[EEXIST] = TEXT_EEXIST;
4630032     err[EFAULT] = TEXT_EFAULT;
4630033     err[EFBIG] = TEXT_EFBIG;
4630034     err[EHOSTUNREACH] = TEXT_EHOSTUNREACH;
```

```
4630035 err[EIDRM] = TEXT_EIDRM;
4630036 err[EILSEQ] = TEXT_EILSEQ;
4630037 err[EINPROGRESS] = TEXT_EINPROGRESS;
4630038 err[EINTR] = TEXT_EINTR;
4630039 err[EINVAL] = TEXT_EINVAL;
4630040 err[EIO] = TEXT_EIO;
4630041 err[EISCONN] = TEXT_EISCONN;
4630042 err[EISDIR] = TEXT_EISDIR;
4630043 err[ELOOP] = TEXT_ELOOP;
4630044 err[EMFILE] = TEXT_EMFILE;
4630045 err[EMLINK] = TEXT_EMLINK;
4630046 err[EMSGSIZE] = TEXT_EMSGSIZE;
4630047 err[EMULTIHOP] = TEXT_EMULTIHOP;
4630048 err[ENAMETOOLONG] = TEXT_ENAMETOOLONG;
4630049 err[ENETDOWN] = TEXT_ENETDOWN;
4630050 err[ENETRESET] = TEXT_ENETRESET;
4630051 err[ENETUNREACH] = TEXT_ENETUNREACH;
4630052 err[ENFILE] = TEXT_ENFILE;
4630053 err[ENOBUFS] = TEXT_ENOBUFS;
4630054 err[ENODATA] = TEXT_ENODATA;
4630055 err[ENODEV] = TEXT_ENODEV;
4630056 err[ENOENT] = TEXT_ENOENT;
4630057 err[ENOEXEC] = TEXT_ENOEXEC;
4630058 err[ENOLCK] = TEXT_ENOLCK;
4630059 err[ENOLINK] = TEXT_ENOLINK;
4630060 err[ENOMEM] = TEXT_ENOMEM;
4630061 err[ENOMSG] = TEXT_ENOMSG;
4630062 err[ENOPROTOOPT] = TEXT_ENOPROTOOPT;
4630063 err[ENOSPC] = TEXT_ENOSPC;
4630064 err[ENOSR] = TEXT_ENOSR;
4630065 err[ENOSTR] = TEXT_ENOSTR;
4630066 err[ENOSYS] = TEXT_ENOSYS;
4630067 err[ENOTCONN] = TEXT_ENOTCONN;
4630068 err[ENOTDIR] = TEXT_ENOTDIR;
4630069 err[ENOTEMPTY] = TEXT_ENOTEMPTY;
4630070 err[ENOTSOCK] = TEXT_ENOTSOCK;
4630071 err[ENOTSUP] = TEXT_ENOTSUP;
```

```
4630072 err[ENOTTY] = TEXT_ENOTTY;
4630073 err[ENXIO] = TEXT_ENXIO;
4630074 err[EOPNOTSUPP] = TEXT_EOPNOTSUPP;
4630075 err[EOVERFLOW] = TEXT_EOVERFLOW;
4630076 err[EPERM] = TEXT_EPERM;
4630077 err[EPIPE] = TEXT_EPIPE;
4630078 err[EPROTO] = TEXT_EPROTO;
4630079 err[EPROTONOSUPPORT] = TEXT_EPROTONOSUPPORT;
4630080 err[EPROTOTYPE] = TEXT_EPROTOTYPE;
4630081 err[ERANGE] = TEXT_ERANGE;
4630082 err[EROFS] = TEXT_EROFS;
4630083 err[ESPIPE] = TEXT_ESPIPE;
4630084 err[ESRCH] = TEXT_ESRCH;
4630085 err[ESTALE] = TEXT_ESTALE;
4630086 err[ETIME] = TEXT_ETIME;
4630087 err[ETIMEDOUT] = TEXT_ETIMEDOUT;
4630088 err[ETXTBSY] = TEXT_ETXTBSY;
4630089 err[EWOULDBLOCK] = TEXT_EWOULDBLOCK;
4630090 err[EXDEV] = TEXT_EXDEV;
4630091 err[E_NO_MEDIUM] = TEXT_E_NO_MEDIUM;
4630092 err[E_MEDIUM] = TEXT_E_MEDIUM;
4630093 err[E_FILE_TYPE] = TEXT_E_FILE_TYPE;
4630094 err[E_ROOT_INODE_NOT_CACHED] =
4630095     TEXT_E_ROOT_INODE_NOT_CACHED;
4630096 err[E_CANNOT_READ_SUPERBLOCK] =
4630097     TEXT_E_CANNOT_READ_SUPERBLOCK;
4630098 err[E_MAP_INODE_TOO_BIG] = TEXT_E_MAP_INODE_TOO_BIG;
4630099 err[E_MAP_ZONE_TOO_BIG] = TEXT_E_MAP_ZONE_TOO_BIG;
4630100 err[E_DATA_ZONE_TOO_BIG] = TEXT_E_DATA_ZONE_TOO_BIG;
4630101 err[E_CANNOT_FIND_ROOT_DEVICE] =
4630102     TEXT_E_CANNOT_FIND_ROOT_DEVICE;
4630103 err[E_CANNOT_FIND_ROOT_INODE] =
4630104     TEXT_E_CANNOT_FIND_ROOT_INODE;
4630105 err[E_FILE_TYPE_UNSUPPORTED] =
4630106     TEXT_E_FILE_TYPE_UNSUPPORTED;
4630107 err[E_ENV_TOO_BIG] = TEXT_E_ENV_TOO_BIG;
4630108 err[E_LIMIT] = TEXT_E_LIMIT;
```



```
4630109     err[E_NOT_MOUNTED] = TEXT_E_NOT_MOUNTED;
4630110     err[E_NOT_IMPLEMENTED] = TEXT_E_NOT_IMPLEMENTED;
4630111     err[E_HARDWARE_FAULT] = TEXT_E_HARDWARE_FAULT;
4630112     err[E_DRIVER_FAULT] = TEXT_E_DRIVER_FAULT;
4630113     err[E_PIPE_FULL] = TEXT_E_PIPE_FULL;
4630114     err[E_PIPE_EMPTY] = TEXT_E_PIPE_EMPTY;
4630115     err[E_PART_TYPE_NOT_MINIX] = TEXT_E_PART_TYPE_NOT_MINIX;
4630116     err[E_FS_TYPE_NOT_SUPPORTED] =
4630117         TEXT_E_FS_TYPE_NOT_SUPPORTED;
4630118     err[E_PDU_TOO_BIG] = TEXT_E_PDU_TOO_BIG;
4630119     err[E_ARP_MISSING] = TEXT_E_ARP_MISSING;
4630120     //
4630121     if (errno >= ERROR_MAX || errno < 0)
4630122     {
4630123         return ("Unknown error");
4630124     }
4630125     //
4630126     return (err[errno]);
4630127 }
```

95.20.15 lib/string/strlen.c

Si veda la sezione [88.121](#).

```
4640001 #include <string.h>
4640002 //-----
4640003 size_t
4640004 strlen (const char *string)
4640005 {
4640006     size_t i;
4640007     for (i = 0; string[i] != 0; i++)
4640008     {
4640009         ; // Just count.
4640010     }
4640011     return i;
4640012 }
```



95.20.16 lib/string/strncat.c



Si veda la sezione [88.113](#).

```
4650001 #include <string.h>
4650002 //-----
4650003 char *
4650004 strncat (char *restrict dst, const char *restrict org,
4650005         size_t n)
4650006 {
4650007     size_t i;
4650008     size_t j;
4650009     for (i = 0; n > 0 && dst[i] != 0; i++)
4650010     {
4650011         ; // Just seek the null character.
4650012     }
4650013     for (j = 0; n > 0 && j < n && org[j] != 0; i++, j++)
4650014     {
4650015         dst[i] = org[j];
4650016     }
4650017     dst[i] = 0;
4650018     return dst;
4650019 }
```

95.20.17 lib/string/strncmp.c



Si veda la sezione [88.115](#).

```
4660001 #include <string.h>
4660002 //-----
4660003 int
4660004 strncmp (const char *string1, const char *string2, size_t n)
4660005 {
4660006     size_t i;
4660007     for (i = 0; i < n; i++)
4660008     {
4660009         if (string1[i] > string2[i])
4660010         {
```

```
4660011         return 1;
4660012     }
4660013     else if (string1[i] < string2[i])
4660014     {
4660015         return -1;
4660016     }
4660017     else if (string1[i] == 0 && string2[i] == 0)
4660018     {
4660019         return 0;
4660020     }
4660021 }
4660022 return 0;
4660023 }
```

95.20.18 lib/string/strncpy.c

Si veda la sezione [88.117](#).



```
4670001 #include <string.h>
4670002 //-----
4670003 char *
4670004 strncpy (char *restrict dst, const char *restrict org,
4670005         size_t n)
4670006 {
4670007     size_t i;
4670008     for (i = 0; n > 0 && i < n && org[i] != 0; i++)
4670009     {
4670010         dst[i] = org[i];
4670011     }
4670012     for (; n > 0 && i < n; i++)
4670013     {
4670014         dst[i] = 0;
4670015     }
4670016     return dst;
4670017 }
```

95.20.19 lib/string/strpbrk.c



Si veda la sezione [88.125](#).

```
4680001 #include <string.h>
4680002 //-----
4680003 char *
4680004 strpbrk (const char *string, const char *accept)
4680005 {
4680006     //
4680007     // The first parameter not 'const char *' because
4680008     // otherwise
4680009     // the return value should be 'const char *' too!
4680010     //
4680011     size_t i;
4680012     size_t j;
4680013     //
4680014     for (i = 0; string[i] != 0; i++)
4680015     {
4680016         for (j = 0; accept[j] != 0; j++)
4680017         {
4680018             if (string[i] == accept[j])
4680019             {
4680020                 return (char *) (string + i);
4680021             }
4680022         }
4680023     }
4680024     return NULL;
4680025 }
```

95.20.20 lib/string/strchr.c



Si veda la sezione [88.114](#).

```
4690001 #include <string.h>
4690002 //-----
4690003 char *
4690004 strchr (const char *string, int c)
```

```
4690005 {
4690006     int i;
4690007     for (i = strlen (string); i >= 0; i--)
4690008     {
4690009         if (string[i] == (char) c)
4690010         {
4690011             break;
4690012         }
4690013     }
4690014     if (i < 0)
4690015     {
4690016         return NULL;
4690017     }
4690018     else
4690019     {
4690020         return (char *) (string + i);
4690021     }
4690022 }
```

95.20.21 lib/string/strspn.c

Si veda la sezione [88.127](#).

```
4700001 #include <string.h>
4700002 //-----
4700003 size_t
4700004 strspn (const char *string, const char *accept)
4700005 {
4700006     size_t i;
4700007     size_t j;
4700008     int found;
4700009     for (i = 0; string[i] != 0; i++)
4700010     {
4700011         for (j = 0, found = 0; accept[j] != 0; j++)
4700012         {
4700013             if (string[i] == accept[j])
4700014                 {
```



```
4700015         found = 1;
4700016         break;
4700017     }
4700018 }
4700019     if (!found)
4700020     {
4700021         break;
4700022     }
4700023 }
4700024 return i;
4700025 }
```

95.20.22 lib/string/strstr.c



Si veda la sezione [88.128](#).

```
4710001 #include <string.h>
4710002 //-----
4710003 char *
4710004 strstr (const char *string, const char *substring)
4710005 {
4710006     size_t i;
4710007     size_t j;
4710008     size_t k;
4710009     int found;
4710010     if (substring[0] == 0)
4710011     {
4710012         return (char *) string;
4710013     }
4710014     for (i = 0, j = 0, found = 0; string[i] != 0; i++)
4710015     {
4710016         if (string[i] == substring[0])
4710017         {
4710018             for (k = i, j = 0;
4710019                 string[k] == substring[j] &&
4710020                 string[k] != 0 &&
4710021                 substring[j] != 0; j++, k++)
```

```
4710022         {
4710023             ;
4710024         }
4710025         if (substring[j] == 0)
4710026         {
4710027             found = 1;
4710028         }
4710029     }
4710030     if (found)
4710031     {
4710032         return (char *) (string + i);
4710033     }
4710034 }
4710035 return NULL;
4710036 }
```

95.20.23 lib/string/strtok.c

Si veda la sezione [88.129](#).

```
4720001 #include <string.h>
4720002 //-----
4720003 char *
4720004 strtok (char *restrict string, const char *restrict delim)
4720005 {
4720006     static char *next = NULL;
4720007     size_t i = 0;
4720008     size_t j;
4720009     int found_token;
4720010     int found_delim;
4720011     //
4720012     // If the string received a the first parameter is a
4720013     // null pointer,
4720014     // the static pointer is used. But if it is already
4720015     // NULL,
4720016     // the scan cannot start.
4720017     //
```

```
4720018     if (string == NULL)
4720019     {
4720020         if (next == NULL)
4720021         {
4720022             return NULL;
4720023         }
4720024     else
4720025     {
4720026         string = next;
4720027     }
4720028 }
4720029 //
4720030 // If the string received as the first parameter is
4720031 // empty, the scan
4720032 // cannot start.
4720033 //
4720034 if (string[0] == 0)
4720035 {
4720036     next = NULL;
4720037     return NULL;
4720038 }
4720039 else
4720040 {
4720041     if (delim[0] == 0)
4720042     {
4720043         return string;
4720044     }
4720045 }
4720046 //
4720047 // Find the next token.
4720048 //
4720049 for (i = 0, found_token = 0, j = 0;
4720050     string[i] != 0 && (!found_token); i++)
4720051 {
4720052     //
4720053     // Look inside delimiters.
4720054     //
```



```
4720055     for (j = 0, found_delim = 0; delim[j] != 0; j++)
4720056     {
4720057         if (string[i] == delim[j])
4720058         {
4720059             found_delim = 1;
4720060         }
4720061     }
4720062     //
4720063     // If current character inside the string is not
4720064     // a delimiter,
4720065     // it is the start of a new token.
4720066     //
4720067     if (!found_delim)
4720068     {
4720069         found_token = 1;
4720070         break;
4720071     }
4720072 }
4720073 //
4720074 // If a token was found, the pointer is updated.
4720075 // If otherwise the token is not found, this means
4720076 // that
4720077 // there are no more.
4720078 //
4720079 if (found_token)
4720080 {
4720081     string += i;
4720082 }
4720083 else
4720084 {
4720085     next = NULL;
4720086     return NULL;
4720087 }
4720088 //
4720089 // Find the end of the token.
4720090 //
4720091 for (i = 0, found_delim = 0; string[i] != 0; i++)
```

```
4720092     {
4720093         for (j = 0; delim[j] != 0; j++)
4720094             {
4720095                 if (string[i] == delim[j])
4720096                     {
4720097                         found_delim = 1;
4720098                         break;
4720099                     }
4720100             }
4720101         if (found_delim)
4720102             {
4720103                 break;
4720104             }
4720105     }
4720106     //
4720107     // If a delimiter was found, the corresponding
4720108     // character must be
4720109     // reset to zero. If otherwise the string is
4720110     // terminated, the
4720111     // scan is terminated.
4720112     //
4720113     if (found_delim)
4720114         {
4720115             string[i] = 0;
4720116             next = &string[i + 1];
4720117         }
4720118     else
4720119         {
4720120             next = NULL;
4720121         }
4720122     //
4720123     // At this point, the current string represent the
4720124     // token found.
4720125     //
4720126     return string;
4720127 }
```

95.20.24 lib/string/strxfrm.c



Si veda la sezione [88.132](#).

```
4730001 #include <string.h>
4730002 //-----
4730003 size_t
4730004 strxfrm (char *restrict dst, const char *restrict org,
4730005         size_t n)
4730006 {
4730007     size_t i;
4730008     if (n == 0 && dst == NULL)
4730009     {
4730010         return strlen (org);
4730011     }
4730012     else
4730013     {
4730014         for (i = 0; i < n; i++)
4730015         {
4730016             dst[i] = org[i];
4730017             if (org[i] == 0)
4730018             {
4730019                 break;
4730020             }
4730021         }
4730022         return i;
4730023     }
4730024 }
```

95.21 os32: «lib/sys/os32.h»



Si veda la sezione [91.3](#).

```
4740001 #ifndef _SYS_OS32_H
4740002 #define _SYS_OS32_H      1
4740003 //-----
4740004 // This file contains all the declarations that don't
4740005 // have a better place inside standard headers files.
```

```
4740006 // Even declarations related to device numbers and
4740007 // system calls is contained here.
4740008 //-----
4740009 #include <sys/types.h>
4740010 #include <sys/stat.h>
4740011 #include <sys/socket.h>
4740012 #include <arpa/inet.h>
4740013 #include <netinet/in.h>
4740014 #include <stdint.h>
4740015 #include <signal.h>
4740016 #include <limits.h>
4740017 #include <stdio.h>
4740018 #include <stddef.h>
4740019 #include <restrict.h>
4740020 #include <stdarg.h>
4740021 #include <termios.h>
4740022 //-----
4740023 typedef uint16_t h_port_t; // Port number in host
4740024 // byte order.
4740025 typedef uint32_t h_addr_t; // IPv4 address in
4740026 // host byte order.
4740027 //-----
4740028 // Please remember that system calls should never be
4740029 // used (called) inside the kernel code, because system
4740030 // calls cannot be nested for the os32 simple
4740031 // architecture!
4740032 // If a particular function is necessary inside the
4740033 // kernel, that usually is made by a system call, an
4740034 // appropriate k_...() function must be
4740035 // made, to avoid the problem.
4740036 //-----
4740037 // Device numbers.
4740038 //-----
4740039 #define DEV_UNDEFINED_MAJOR ((dev_t) 0x00)
4740040 #define DEV_UNDEFINED ((dev_t) 0x0000)
4740041 #define DEV_MEM_MAJOR ((dev_t) 0x01)
4740042 #define DEV_MEM ((dev_t) 0x0101)
```

```
4740043 #define DEV_NULL ((dev_t) 0x0102)
4740044 #define DEV_PORT ((dev_t) 0x0103)
4740045 #define DEV_ZERO ((dev_t) 0x0104)
4740046 #define DEV_TTY_MAJOR ((dev_t) 0x02)
4740047 #define DEV_TTY ((dev_t) 0x0200)
4740048 //
4740049 #define DEV_KMEM_MAJOR ((dev_t) 0x04)
4740050 #define DEV_KMEM_PS ((dev_t) 0x0401)
4740051 #define DEV_KMEM_MMP ((dev_t) 0x0402)
4740052 #define DEV_KMEM_SB ((dev_t) 0x0403)
4740053 #define DEV_KMEM_INODE ((dev_t) 0x0404)
4740054 #define DEV_KMEM_FILE ((dev_t) 0x0405)
4740055 #define DEV_KMEM_ARP ((dev_t) 0x0406)
4740056 #define DEV_KMEM_NET ((dev_t) 0x0407)
4740057 #define DEV_KMEM_ROUTE ((dev_t) 0x0408)
4740058 //
4740059 #define DEV_CONSOLE_MAJOR ((dev_t) 0x05)
4740060 #define DEV_CONSOLE ((dev_t) 0x05FF)
4740061 #define DEV_CONSOLE0 ((dev_t) 0x0500)
4740062 #define DEV_CONSOLE1 ((dev_t) 0x0501)
4740063 #define DEV_CONSOLE2 ((dev_t) 0x0502)
4740064 #define DEV_CONSOLE3 ((dev_t) 0x0503)
4740065 #define DEV_CONSOLE4 ((dev_t) 0x0504)
4740066 //
4740067 #define DEV_DM_MAJOR ((dev_t) 0x08)
4740068 #define DEV_DM00 ((dev_t) 0x0800)
4740069 #define DEV_DM01 ((dev_t) 0x0801)
4740070 #define DEV_DM02 ((dev_t) 0x0802)
4740071 #define DEV_DM03 ((dev_t) 0x0803)
4740072 #define DEV_DM04 ((dev_t) 0x0804)
4740073 #define DEV_DM10 ((dev_t) 0x0810)
4740074 #define DEV_DM11 ((dev_t) 0x0811)
4740075 #define DEV_DM12 ((dev_t) 0x0812)
4740076 #define DEV_DM13 ((dev_t) 0x0813)
4740077 #define DEV_DM14 ((dev_t) 0x0814)
4740078 #define DEV_DM20 ((dev_t) 0x0820)
4740079 #define DEV_DM21 ((dev_t) 0x0821)
```

```
4740080 #define DEV_DM22 ((dev_t) 0x0822)
4740081 #define DEV_DM23 ((dev_t) 0x0823)
4740082 #define DEV_DM24 ((dev_t) 0x0824)
4740083 #define DEV_DM30 ((dev_t) 0x0830)
4740084 #define DEV_DM31 ((dev_t) 0x0831)
4740085 #define DEV_DM32 ((dev_t) 0x0832)
4740086 #define DEV_DM33 ((dev_t) 0x0833)
4740087 #define DEV_DM34 ((dev_t) 0x0834)
4740088 //
4740089 //-----
4740090 #define min(a, b) (a < b ? a : b)
4740091 #define max(a, b) (a > b ? a : b)
4740092 #define sizeof_array(x) (sizeof(x) / sizeof((x)[0]))
4740093 #define sizeof_field(t, f) (sizeof(((t*)0)->f))
4740094 //-----
4740095 #define INPUT_LINE_HIDDEN 0
4740096 #define INPUT_LINE_ECHO 1
4740097 //-----
4740098 #define MOUNT_DEFAULT 0 // Default mount
4740099 // options.
4740100 #define MOUNT_RO 1 // Read only mount
4740101 // option.
4740102 //-----
4740103 #define SYS_0 0 // Nothing to
4740104 // do.
4740105 #define SYS_CHDIR 1
4740106 #define SYS_CHMOD 2
4740107 #define SYS_CLOCK 3
4740108 #define SYS_CLOSE 4
4740109 #define SYS_EXEC 5
4740110 #define SYS_EXIT 6 // [1] see
4740111 // below.
4740112 #define SYS_FCHMOD 7
4740113 #define SYS_FORK 8
4740114 #define SYS_FSTAT 9
4740115 #define SYS_KILL 10
4740116 #define SYS_LSEEK 11
```

4740117	#define SYS_MKDIR	12	
4740118	#define SYS_MKNOD	13	
4740119	#define SYS_MOUNT	14	
4740120	#define SYS_OPEN	15	
4740121	#define SYS_PGRP	16	
4740122	#define SYS_READ	17	
4740123	#define SYS_SETEUID	18	
4740124	#define SYS_SETUID	19	
4740125	#define SYS_SIGNAL	20	
4740126	#define SYS_SLEEP	21	
4740127	#define SYS_STAT	22	
4740128	#define SYS_TIME	23	
4740129	#define SYS_UAREA	24	
4740130	#define SYS_UMASK	25	
4740131	#define SYS_UMOUNT	26	
4740132	#define SYS_WAIT	27	
4740133	#define SYS_WRITE	28	
4740134	#define SYS_ZPCHAR	29	// [2]
4740135	#define SYS_ZPSTRING	30	// [2]
4740136	#define SYS_CHOWN	31	
4740137	#define SYS_DUP	33	
4740138	#define SYS_DUP2	34	
4740139	#define SYS_LINK	35	
4740140	#define SYS_UNLINK	36	
4740141	#define SYS_FCNTL	37	
4740142	#define SYS_STIME	38	
4740143	#define SYS_FCHOWN	39	
4740144	#define SYS_BRK	40	
4740145	#define SYS_SBRK	41	
4740146	#define SYS_PIPE	42	
4740147	#define SYS_TCGETATTR	43	
4740148	#define SYS_TCSETATTR	44	
4740149	#define SYS_SETEGID	45	
4740150	#define SYS_SETGID	46	
4740151	#define SYS_SETJMP	47	
4740152	#define SYS_LONGJMP	48	
4740153	#define SYS_RECVFROM	49	

```
4740154 #define SYS_SOCKET 50
4740155 #define SYS_CONNECT 51
4740156 #define SYS_SEND 52
4740157 #define SYS_IPCONFIG 53
4740158 #define SYS_ROUTEADD 54
4740159 #define SYS_ROUTEDEL 55
4740160 #define SYS_BIND 56
4740161 #define SYS_LISTEN 57
4740162 #define SYS_ACCEPT 58
4740163 //
4740164 // [1] The files 'crt0...' need to know the value used
4740165 // for the exit system call. If this value is
4740166 // modified, all the file 'crt0...' have also to be
4740167 // modified the same way.
4740168 //
4740169 // [2] These system calls were developed at the
4740170 // beginning, when no standard I/O was available.
4740171 // They are to be considered as a last resort for
4740172 // debugging purposes.
4740173 //
4740174 //-----
4740175 // The following values must be: 1, 2, 4, 8, 16, 32,...
4740176 // so that can be 'OR' combined.
4740177 //
4740178 #define WAKEUP_EVENT_SIGNAL 0x0001
4740179 #define WAKEUP_EVENT_TIMER 0x0002
4740180 #define WAKEUP_EVENT_DEV_READ 0x0004
4740181 #define WAKEUP_EVENT_DEV_WRITE 0x0008
4740182 #define WAKEUP_EVENT_PIPE_READ 0x0010
4740183 #define WAKEUP_EVENT_PIPE_WRITE 0x0020
4740184 #define WAKEUP_EVENT SOCK_READ 0x0040
4740185 #define WAKEUP_EVENT SOCK_WRITE 0x0080
4740186 //-----
4740187 typedef struct
4740188 {
4740189     int sfdn;
4740190     struct sockaddr addr;
```



```
4740191     socklen_t  addrlen;
4740192     int  fl_flags;
4740193     int  ret;
4740194     int  errno;
4740195     int  errln;
4740196     char  errfn[PATH_MAX];
4740197 } sysmsg_accept_t;
4740198 //-----
4740199 typedef struct
4740200 {
4740201     int  sfdn;
4740202     struct  sockaddr  addr;
4740203     socklen_t  addrlen;
4740204     int  ret;
4740205     int  errno;
4740206     int  errln;
4740207     char  errfn[PATH_MAX];
4740208 } sysmsg_bind_t;
4740209 //-----
4740210 typedef struct
4740211 {
4740212     void *address;
4740213     int  ret;
4740214     int  errno;
4740215     int  errln;
4740216     char  errfn[PATH_MAX];
4740217 } sysmsg_brk_t;
4740218 //-----
4740219 typedef struct
4740220 {
4740221     const char *path;
4740222     int  ret;
4740223     int  errno;
4740224     int  errln;
4740225     char  errfn[PATH_MAX];
4740226 } sysmsg_chdir_t;
4740227 //-----
```

```
4740228 typedef struct
4740229 {
4740230     const char *path;
4740231     mode_t mode;
4740232     int ret;
4740233     int errno;
4740234     int errln;
4740235     char errfn[PATH_MAX];
4740236 } sysmsg_chmod_t;
4740237 //-----
4740238 typedef struct
4740239 {
4740240     const char *path;
4740241     uid_t uid;
4740242     uid_t gid;
4740243     int ret;
4740244     int errno;
4740245     int errln;
4740246     char errfn[PATH_MAX];
4740247 } sysmsg_chown_t;
4740248 //-----
4740249 typedef struct
4740250 {
4740251     clock_t ret;
4740252 } sysmsg_clock_t;
4740253 //-----
4740254 typedef struct
4740255 {
4740256     int fdn;
4740257     int ret;
4740258     int errno;
4740259     int errln;
4740260     char errfn[PATH_MAX];
4740261 } sysmsg_close_t;
4740262 //-----
4740263 typedef struct
4740264 {
```

```
4740265     int sfdn;
4740266     struct sockaddr addr;
4740267     socklen_t addrlen;
4740268     int ret;
4740269     int errno;
4740270     int errln;
4740271     char errfn[PATH_MAX];
4740272 } sysmsg_connect_t;
4740273 //-----
4740274 typedef struct
4740275 {
4740276     int fdn_old;
4740277     int ret;
4740278     int errno;
4740279     int errln;
4740280     char errfn[PATH_MAX];
4740281 } sysmsg_dup_t;
4740282 //-----
4740283 typedef struct
4740284 {
4740285     int fdn_old;
4740286     int fdn_new;
4740287     int ret;
4740288     int errno;
4740289     int errln;
4740290     char errfn[PATH_MAX];
4740291 } sysmsg_dup2_t;
4740292 //-----
4740293 typedef struct
4740294 {
4740295     const char *path;
4740296     int argc;
4740297     int envc;
4740298     char arg_data[ARG_MAX / 2];
4740299     char env_data[ARG_MAX / 2];
4740300     uid_t uid;
4740301     uid_t euid;
```

```
4740302     int ret;
4740303     int errno;
4740304     int errln;
4740305     char errfn[PATH_MAX];
4740306 } sysmsg_exec_t;
4740307 //-----
4740308 typedef struct
4740309 {
4740310     int status;
4740311 } sysmsg_exit_t;
4740312 //-----
4740313 typedef struct
4740314 {
4740315     int fdn;
4740316     mode_t mode;
4740317     int ret;
4740318     int errno;
4740319     int errln;
4740320     char errfn[PATH_MAX];
4740321 } sysmsg_fchmod_t;
4740322 //-----
4740323 typedef struct
4740324 {
4740325     int fdn;
4740326     uid_t uid;
4740327     uid_t gid;
4740328     int ret;
4740329     int errno;
4740330     int errln;
4740331     char errfn[PATH_MAX];
4740332 } sysmsg_fchown_t;
4740333 //-----
4740334 typedef struct
4740335 {
4740336     int fdn;
4740337     int cmd;
4740338     int arg;
```

```
4740339     int ret;
4740340     int errno;
4740341     int errln;
4740342     char errfn[PATH_MAX];
4740343 } sysmsg_fcntl_t;
4740344 //-----
4740345 typedef struct
4740346 {
4740347     pid_t ret;
4740348     int errno;
4740349     int errln;
4740350     char errfn[PATH_MAX];
4740351 } sysmsg_fork_t;
4740352 //-----
4740353 typedef struct
4740354 {
4740355     int fdn;
4740356     struct stat stat;
4740357     int ret;
4740358     int errno;
4740359     int errln;
4740360     char errfn[PATH_MAX];
4740361 } sysmsg_fstat_t;
4740362 //-----
4740363 typedef struct
4740364 {
4740365     int n;
4740366     in_addr_t address;
4740367     int m;
4740368     int ret;
4740369     int errno;
4740370     int errln;
4740371     char errfn[PATH_MAX];
4740372 } sysmsg_ipconfig_t;
4740373 //-----
4740374 typedef struct
4740375 {
```

```
4740376     void *env;
4740377     int ret;
4740378     //
4740379     // This structure is intentionally reduced.
4740380     //
4740381 } sysmsg_jump_t;
4740382 //-----
4740383 typedef struct
4740384 {
4740385     pid_t pid;
4740386     int signal;
4740387     int ret;
4740388     int errno;
4740389     int errln;
4740390     char errfn[PATH_MAX];
4740391 } sysmsg_kill_t;
4740392 //-----
4740393 typedef struct
4740394 {
4740395     const char *path_old;
4740396     const char *path_new;
4740397     int ret;
4740398     int errno;
4740399     int errln;
4740400     char errfn[PATH_MAX];
4740401 } sysmsg_link_t;
4740402 //-----
4740403 typedef struct
4740404 {
4740405     int sfdn;
4740406     int backlog;
4740407     int ret;
4740408     int errno;
4740409     int errln;
4740410     char errfn[PATH_MAX];
4740411 } sysmsg_listen_t;
4740412 //-----
```

```
4740413 typedef struct
4740414 {
4740415     int fdn;
4740416     off_t offset;
4740417     int whence;
4740418     int ret;
4740419     int errno;
4740420     int errln;
4740421     char errfn[PATH_MAX];
4740422 } sysmsg_lseek_t;
4740423 //-----
4740424 typedef struct
4740425 {
4740426     const char *path;
4740427     mode_t mode;
4740428     int ret;
4740429     int errno;
4740430     int errln;
4740431     char errfn[PATH_MAX];
4740432 } sysmsg_mkdir_t;
4740433 //-----
4740434 typedef struct
4740435 {
4740436     const char *path;
4740437     mode_t mode;
4740438     dev_t device;
4740439     int ret;
4740440     int errno;
4740441     int errln;
4740442     char errfn[PATH_MAX];
4740443 } sysmsg_mknod_t;
4740444 //-----
4740445 typedef struct
4740446 {
4740447     const char *path_dev;
4740448     const char *path_mnt;
4740449     int options;
```

```
4740450     int ret;
4740451     int errno;
4740452     int errln;
4740453     char errfn[PATH_MAX];
4740454 } sysmsg_mount_t;
4740455 //-----
4740456 typedef struct
4740457 {
4740458     const char *path;
4740459     int flags;
4740460     mode_t mode;
4740461     int ret;
4740462     int errno;
4740463     int errln;
4740464     char errfn[PATH_MAX];
4740465 } sysmsg_open_t;
4740466 //-----
4740467 typedef struct
4740468 {
4740469     int pipefd[2];
4740470     int ret;
4740471     int errno;
4740472     int errln;
4740473     char errfn[PATH_MAX];
4740474 } sysmsg_pipe_t;
4740475 //-----
4740476 typedef struct
4740477 {
4740478     int fdn;
4740479     void *buffer;
4740480     size_t count;
4740481     int fl_flags;
4740482     ssize_t ret;
4740483     int errno;
4740484     int errln;
4740485     char errfn[PATH_MAX];
4740486 } sysmsg_read_t;
```



```
4740487 //-----
4740488 typedef struct
4740489 {
4740490     int sfdn;
4740491     void *buffer;
4740492     size_t count;
4740493     int flags;
4740494     void *addrfrom;
4740495     void *addrsz;
4740496     int fl_flags;
4740497     ssize_t ret;
4740498     int errno;
4740499     int errln;
4740500     char errfn[PATH_MAX];
4740501 } sysmsg_recvfrom_t;
4740502 //-----
4740503 typedef struct
4740504 {
4740505     in_addr_t destination;
4740506     int m;
4740507     in_addr_t router;
4740508     int device;
4740509     int ret;
4740510     int errno;
4740511     int errln;
4740512     char errfn[PATH_MAX];
4740513 } sysmsg_route_t;
4740514 //-----
4740515 typedef struct
4740516 {
4740517     intptr_t increment;
4740518     void *ret;
4740519     int errno;
4740520     int errln;
4740521     char errfn[PATH_MAX];
4740522 } sysmsg_sbrk_t;
4740523 //-----
```

```
4740524 typedef struct
4740525 {
4740526     int sfdn;
4740527     const void *buffer;
4740528     size_t count;
4740529     int flags;
4740530     ssize_t ret;
4740531     int errno;
4740532     int errln;
4740533     char errfn[PATH_MAX];
4740534 } sysmsg_send_t;
4740535 //-----
4740536 typedef struct
4740537 {
4740538     gid_t egid;
4740539     int ret;
4740540     int errno;
4740541     int errln;
4740542     char errfn[PATH_MAX];
4740543 } sysmsg_setegid_t;
4740544 //-----
4740545 typedef struct
4740546 {
4740547     uid_t euid;
4740548     int ret;
4740549     int errno;
4740550     int errln;
4740551     char errfn[PATH_MAX];
4740552 } sysmsg_seteuid_t;
4740553 //-----
4740554 typedef struct
4740555 {
4740556     gid_t gid;
4740557     gid_t egid;
4740558     gid_t sgid;
4740559     int ret;
4740560     int errno;
```

```
4740561     int errln;
4740562     char errfn[PATH_MAX];
4740563 } sysmsg_setgid_t;
4740564 //-----
4740565 typedef struct
4740566 {
4740567     uid_t uid;
4740568     uid_t euid;
4740569     uid_t suid;
4740570     int ret;
4740571     int errno;
4740572     int errln;
4740573     char errfn[PATH_MAX];
4740574 } sysmsg_setuid_t;
4740575 //-----
4740576 typedef struct
4740577 {
4740578     uintptr_t wrapper;
4740579     sighandler_t handler;
4740580     int signal;
4740581     sighandler_t ret;
4740582     int errno;
4740583     int errln;
4740584     char errfn[PATH_MAX];
4740585 } sysmsg_signal_t;
4740586 //-----
4740587 typedef struct
4740588 {
4740589     int family;
4740590     int type;
4740591     int protocol;
4740592     int ret;
4740593     int errno;
4740594     int errln;
4740595     char errfn[PATH_MAX];
4740596 } sysmsg_socket_t;
4740597 //-----
```

```
4740598 typedef struct
4740599 {
4740600     int events;
4740601     int signal;
4740602     unsigned int seconds;
4740603     time_t ret;
4740604 } sysmsg_sleep_t;
4740605 //-----
4740606 typedef struct
4740607 {
4740608     const char *path;
4740609     struct stat stat;
4740610     int ret;
4740611     int errno;
4740612     int errln;
4740613     char errfn[PATH_MAX];
4740614 } sysmsg_stat_t;
4740615 //-----
4740616 typedef struct
4740617 {
4740618     time_t ret;
4740619 } sysmsg_time_t;
4740620 //-----
4740621 typedef struct
4740622 {
4740623     time_t timer;
4740624     int ret;
4740625 } sysmsg_stime_t;
4740626 //-----
4740627 typedef struct
4740628 {
4740629     int fdn;
4740630     int action;
4740631     struct termios *attr;
4740632     int ret;
4740633     int errno;
4740634     int errln;
```

```
4740635     char errfn[PATH_MAX];
4740636 } sysmsg_tcattrib_t;
4740637 //-----
4740638 typedef struct
4740639 {
4740640     uid_t uid;      // Read user ID.
4740641     uid_t euid;    // Effective user ID.
4740642     uid_t suid;    // Saved user ID.
4740643     gid_t gid;    // Read group ID.
4740644     gid_t egid;   // Effective group ID.
4740645     gid_t sgid;   // Saved group ID.
4740646     pid_t pid;    // Process ID.
4740647     pid_t ppid;   // Parent PID.
4740648     pid_t pgrp;   // Process group.
4740649     mode_t umask; // Access permission mask.
4740650     char *path_cwd;
4740651     size_t path_cwd_size; // Max path size.
4740652 } sysmsg_uarea_t;
4740653 //-----
4740654 typedef struct
4740655 {
4740656     mode_t umask;
4740657     mode_t ret;
4740658 } sysmsg_umask_t;
4740659 //-----
4740660 typedef struct
4740661 {
4740662     const char *path_mnt;
4740663     int ret;
4740664     int errno;
4740665     int errln;
4740666     char errfn[PATH_MAX];
4740667 } sysmsg_umount_t;
4740668 //-----
4740669 typedef struct
4740670 {
4740671     const char *path;
```

```
4740672     int ret;
4740673     int errno;
4740674     int errln;
4740675     char errfn[PATH_MAX];
4740676 } sysmsg_unlink_t;
4740677 //-----
4740678 typedef struct
4740679 {
4740680     int status;
4740681     pid_t ret;
4740682     int errno;
4740683     int errln;
4740684     char errfn[PATH_MAX];
4740685 } sysmsg_wait_t;
4740686 //-----
4740687 typedef struct
4740688 {
4740689     int fdn;
4740690     const void *buffer;
4740691     size_t count;
4740692     ssize_t ret;
4740693     int errno;
4740694     int errln;
4740695     char errfn[PATH_MAX];
4740696 } sysmsg_write_t;
4740697 //-----
4740698 typedef struct
4740699 {
4740700     char c;
4740701 } sysmsg_zpchar_t;
4740702 //-----
4740703 typedef struct
4740704 {
4740705     char string[BUFSIZ];
4740706 } sysmsg_zpstring_t;
4740707 //-----
4740708 void input_line (char *line, char *prompt, size_t size,
```

```

4740709         int type);
4740710 int mount (const char *path_dev, const char *path_mnt,
4740711         int options);
4740712 int namep (const char *name, char *path, size_t size);
4740713 void sys (int syscallnr, void *message, size_t size);
4740714 int umount (const char *path_mnt);
4740715 void z_perror (const char *string);
4740716 int z_printf (const char *restrict format, ...);
4740717 int z_vprintf (const char *restrict format, va_list arg);
4740718 int ipconfig (int n, h_addr_t address, int m);
4740719 int routedel (h_addr_t destination, int m);
4740720 int routeadd (h_addr_t destination, int m,
4740721             h_addr_t router, int device);
4740722 //-----
4740723 #endif

```

95.21.1	lib/sys/os32/input_line.c	2112
95.21.2	lib/sys/os32/ipconfig.c	2116
95.21.3	lib/sys/os32/mount.c	2117
95.21.4	lib/sys/os32/namep.c	2118
95.21.5	lib/sys/os32/routeadd.c	2122
95.21.6	lib/sys/os32/routedel.c	2124
95.21.7	lib/sys/os32/sys.s	2125
95.21.8	lib/sys/os32/umount.c	2125
95.21.9	lib/sys/os32/z_perror.c	2126
95.21.10	lib/sys/os32/z_printf.c	2127
95.21.11	lib/sys/os32/z_vprintf.c	2128

95.21.1 lib/sys/os32/input_line.c



Si veda la sezione [88.68](#).

```
4750001 #include <sys/os32.h>
4750002 #include <string.h>
4750003 #include <stdio.h>
4750004 #include <errno.h>
4750005 #include <unistd.h>
4750006 //-----
4750007 static int terminal_echo (struct termios *orig);
4750008 static int terminal_noecho (struct termios *orig);
4750009 static int terminal_restore (struct termios *orig);
4750010 //-----
4750011 void
4750012 input_line (char *line, char *prompt, size_t size, int type)
4750013 {
4750014     void *pstatus;
4750015     int i;
4750016     struct termios attr;
4750017     //
4750018     // Set terminal configuration.
4750019     //
4750020     if (type == INPUT_LINE_HIDDEN)
4750021     {
4750022         terminal_noecho (&attr);
4750023     }
4750024     else
4750025     {
4750026         terminal_echo (&attr);
4750027     }
4750028     //
4750029     if (prompt != NULL || strlen (prompt) > 0)
4750030     {
4750031         printf ("%s", prompt);
4750032     }
4750033     //
4750034     errno = 0;
```



```
4750035 pstatus = fgets (line, (int) size, stdin);
4750036 if (pstatus == NULL)
4750037     {
4750038         if (errno)
4750039             {
4750040                 perror (NULL);
4750041             }
4750042         line[0] = 0;
4750043         //
4750044         // Reset terminal mode.
4750045         //
4750046         terminal_restore (&attr);
4750047         return;
4750048     }
4750049     //
4750050     // Find the last position and, if there is a new
4750051     // line code,
4750052     // replace it with zero. If the string is empty, a
4750053     // ^D was
4750054     // received.
4750055     //
4750056     i = strlen (line);
4750057     if (i > 0 && line[i - 1] == '\n')
4750058         {
4750059             line[i - 1] = '\0';
4750060         }
4750061     //
4750062     // Restore terminal mode.
4750063     //
4750064     terminal_restore (&attr);
4750065 }
4750066
4750067 //-----
4750068 static int
4750069 terminal_echo (struct termios *orig)
4750070 {
4750071     int status;
```

```
4750072 struct termios attr;
4750073 //
4750074 // Save previous.
4750075 //
4750076 status = tcgetattr (STDIN_FILENO, orig);
4750077 if (status < 0)
4750078     {
4750079         return (-1);
4750080     }
4750081 //
4750082 // Get again.
4750083 //
4750084 status = tcgetattr (STDIN_FILENO, &attr);
4750085 if (status < 0)
4750086     {
4750087         return (-1);
4750088     }
4750089 //
4750090 attr.c_iflag |= (BRKINT | ICRNL);
4750091 attr.c_iflag &= ~(IGNBRK | INLCR);
4750092 //
4750093 attr.c_lflag |=
4750094     (ECHO | ECHOE | ECHOK | ECHONL | ICANON | ISIG);
4750095 attr.c_lflag &= ~(IEXTEN);
4750096 //
4750097 status = tcsetattr (STDIN_FILENO, TCSANOW, &attr);
4750098 //
4750099 return (status);
4750100 }
4750101
4750102 //-----
4750103 static int
4750104 terminal_noecho (struct termios *orig)
4750105 {
4750106     int status;
4750107     struct termios attr;
4750108     //
```

```
4750109 // Save previous.
4750110 //
4750111 status = tcgetattr (STDIN_FILENO, orig);
4750112 if (status < 0)
4750113     {
4750114         return (-1);
4750115     }
4750116 //
4750117 // Get again.
4750118 //
4750119 status = tcgetattr (STDIN_FILENO, &attr);
4750120 if (status < 0)
4750121     {
4750122         return (-1);
4750123     }
4750124 //
4750125 attr.c_iflag |= (BRKINT | ICRNL);
4750126 attr.c_iflag &= ~(IGNBRK | INLCR);
4750127 //
4750128 attr.c_lflag |= (ICANON | ISIG);
4750129 attr.c_lflag &= ~(ECHO | IEXTEN);
4750130 //
4750131 status = tcsetattr (STDIN_FILENO, TCSANOW, &attr);
4750132 //
4750133 return (status);
4750134 }
4750135
4750136 //-----
4750137 static int
4750138 terminal_restore (struct termios *orig)
4750139 {
4750140     int status;
4750141     //
4750142     // For an unknown reason, when running with Bochs,
4750143     // before
4750144     // restoring the termios configuration, the previous
4750145     // one
```

```
4750146 // is to be read. Here, 'attr' is just a placeholder
4750147 // and
4750148 // the updated content is not used for anything
4750149 // else.
4750150 //
4750151 struct termios attr;
4750152 status = tcgetattr (STDIN_FILENO, &attr);
4750153 if (status < 0)
4750154     {
4750155         return (-1);
4750156     }
4750157 //
4750158 //
4750159 //
4750160 status = tcsetattr (STDIN_FILENO, TCSANOW, orig);
4750161 //
4750162 return (status);
4750163 }
```

95.21.2 lib/sys/os32/ipconfig.c



Si veda la sezione [87.28](#).

```
4760001 #include <sys/os32.h>
4760002 #include <errno.h>
4760003 #include <string.h>
4760004 #include <stdio.h>
4760005 //-----
4760006 int
4760007 ipconfig (int n, in_addr_t address, int m)
4760008 {
4760009     sysmsg_ipconfig_t msg;
4760010     //
4760011     // Fill the message.
4760012     //
4760013     msg.n = n;
4760014     msg.address = address;
```

```

4760015     msg.m = m;
4760016     msg.ret = 0;
4760017     //
4760018     // Syscall.
4760019     //
4760020     sys (SYS_IPCONFIG, &msg, (sizeof msg));
4760021     //
4760022     // Check return value.
4760023     //
4760024     if (msg.ret < 0)
4760025     {
4760026         //
4760027         // Something wrong.
4760028         //
4760029         errno = msg.errno;
4760030         errln = msg.errln;
4760031         strncpy (errfn, msg.errfn, PATH_MAX);
4760032     }
4760033     //
4760034     // Return.
4760035     //
4760036     return (msg.ret);
4760037 }

```

95.21.3 lib/sys/os32/mount.c

Si veda la sezione [87.36](#).

```

4770001     #include <sys/types.h>
4770002     #include <errno.h>
4770003     #include <sys/os32.h>
4770004     #include <stddef.h>
4770005     #include <string.h>
4770006     //-----
4770007     int
4770008     mount (const char *path_dev, const char *path_mnt,
4770009           int options)

```

```
4770010 {
4770011     sysmsg_mount_t msg;
4770012     //
4770013     msg.path_dev = path_dev;
4770014     msg.path_mnt = path_mnt;
4770015     msg.options = options;
4770016     msg.ret = 0;
4770017     msg.errno = 0;
4770018     //
4770019     sys (SYS_MOUNT, &msg, (sizeof msg));
4770020     //
4770021     errno = msg.errno;
4770022     errln = msg.errln;
4770023     strncpy (errfn, msg.errfn, PATH_MAX);
4770024     return (msg.ret);
4770025 }
```

95.21.4 lib/sys/os32/namep.c



Si veda la sezione [88.85](#).

```
4780001 #include <sys/os32.h>
4780002 #include <stdlib.h>
4780003 #include <errno.h>
4780004 #include <unistd.h>
4780005 //-----
4780006 int
4780007 namep (const char *name, char *path, size_t size)
4780008 {
4780009     char command[PATH_MAX];
4780010     char *env_path;
4780011     int p;          // Index used inside the path
4780012     // environment.
4780013     int c;          // Index used inside the command
4780014     // string.
4780015     int status;
4780016     //
```

```
4780017 // Check for valid input.
4780018 //
4780019 if (name == NULL || name[0] == 0 || path == NULL
4780020     || name == path)
4780021     {
4780022         errset (EINVAL); // Invalid argument.
4780023         return (-1);
4780024     }
4780025 //
4780026 // Check if the original command contains at least a
4780027 // '/'. Otherwise
4780028 // a scan for the environment variable 'PATH' must
4780029 // be done.
4780030 //
4780031 if (strchr (name, '/') == NULL)
4780032     {
4780033         //
4780034         // Ok: no '/' there. Get the environment
4780035         // variable 'PATH'.
4780036         //
4780037         env_path = getenv ("PATH");
4780038         if (env_path == NULL)
4780039             {
4780040                 //
4780041                 // There is no 'PATH' environment value.
4780042                 //
4780043                 errset (ENOENT); // No such file or
4780044                 // directory.
4780045                 return (-1);
4780046             }
4780047         //
4780048         // Scan paths and try to find a file with that
4780049         // name.
4780050         //
4780051         for (p = 0; env_path[p] != 0;)
4780052             {
4780053                 for (c = 0;
```

```
4780054         c < (PATH_MAX - strlen (name) - 2) &&
4780055         env_path[p] != 0 &&
4780056         env_path[p] != ':'; c++, p++)
4780057     {
4780058         command[c] = env_path[p];
4780059     }
4780060     //
4780061     // If the loop is ended because the command
4780062     // array does not
4780063     // have enough room for the full path, then
4780064     // must return an
4780065     // error.
4780066     //
4780067     if (env_path[p] != ':' && env_path[p] != 0)
4780068     {
4780069         errset (ENAMETOOLONG);    // Filename
4780070         // too long.
4780071         return (-1);
4780072     }
4780073     //
4780074     // The command array has enough space. At
4780075     // index 'c' must
4780076     // place a zero, to terminate current
4780077     // string.
4780078     //
4780079     command[c] = 0;
4780080     //
4780081     // Add the rest of the path.
4780082     //
4780083     strcat (command, "/");
4780084     strcat (command, name);
4780085     //
4780086     // Verify to have something with that full
4780087     // path name.
4780088     //
4780089     status = access (command, F_OK);
4780090     if (status == 0)
```



```
4780091     {
4780092         //
4780093         // Verify to have enough room inside the
4780094         // destination
4780095         // path.
4780096         //
4780097         if (strlen (command) >= size)
4780098             {
4780099                 //
4780100                 // Sorry: too big. There must be
4780101                 // room also for
4780102                 // the string termination null
4780103                 // character.
4780104                 //
4780105                 errset (ENAMETOOLONG);           // Filename
4780106                 // too long.
4780107                 return (-1);
4780108             }
4780109         //
4780110         // Copy the path and return.
4780111         //
4780112         strncpy (path, command, size);
4780113         return (0);
4780114     }
4780115     //
4780116     // That path was not good: try again. But
4780117     // before returning
4780118     // to the external loop, must verify if 'p'
4780119     // is to be
4780120     // incremented, after a ':', because the
4780121     // external loop
4780122     // does not touch the index 'p',
4780123     //
4780124     if (env_path[p] == ':')
4780125         {
4780126             p++;
4780127         }
```

```
4780128     }
4780129     //
4780130     // At this point, there is no match with the
4780131     // paths.
4780132     //
4780133     errset (ENOENT); // No such file or directory.
4780134     return (-1);
4780135 }
4780136 //
4780137 // At this point, a path was given and the
4780138 // environment variable
4780139 // 'PATH' was not scanned. Just copy the same path.
4780140 // But must verify
4780141 // that the receiving path has enough room for it.
4780142 //
4780143 if (strlen (name) >= size)
4780144 {
4780145     //
4780146     // Sorry: too big.
4780147     //
4780148     errset (ENAMETOOLONG); // Filename too long.
4780149     return (-1);
4780150 }
4780151 //
4780152 // Ok: copy and return.
4780153 //
4780154 strncpy (path, name, size);
4780155 return (0);
4780156 }
```

95.21.5 lib/sys/os32/routeadd.c



Si veda la sezione [87.42](#).

```
4790001 #include <sys/os32.h>
4790002 #include <errno.h>
4790003 #include <string.h>
```

```
4790004 #include <stdio.h>
4790005 //-----
4790006 int
4790007 routeadd (in_addr_t destination, int m,
4790008           in_addr_t router, int device)
4790009 {
4790010     sysmsg_route_t msg;
4790011     //
4790012     // Fill the message.
4790013     //
4790014     msg.destination = destination;
4790015     msg.m = m;
4790016     msg.router = router;
4790017     msg.device = device;
4790018     //
4790019     // Syscall.
4790020     //
4790021     sys (SYS_ROUTEADD, &msg, (sizeof msg));
4790022     //
4790023     // Check return value.
4790024     //
4790025     if (msg.ret < 0)
4790026     {
4790027         //
4790028         // Something wrong.
4790029         //
4790030         errno = msg.errno;
4790031         errln = msg.errln;
4790032         strncpy (errfn, msg.errfn, PATH_MAX);
4790033     }
4790034     //
4790035     // Return.
4790036     //
4790037     return (msg.ret);
4790038 }
```

95.21.6 lib/sys/os32/routedel.c



Si veda la sezione [87.43](#).

```
4800001 #include <sys/os32.h>
4800002 #include <errno.h>
4800003 #include <string.h>
4800004 #include <stdio.h>
4800005 //-----
4800006 int
4800007 routedel (in_addr_t destination, int m)
4800008 {
4800009     sysmsg_route_t msg;
4800010     //
4800011     // Fill the message.
4800012     //
4800013     msg.destination = destination;
4800014     msg.m = m;
4800015     //
4800016     // Syscall.
4800017     //
4800018     sys (SYS_ROUTEDEL, &msg, (sizeof msg));
4800019     //
4800020     // Check return value.
4800021     //
4800022     if (msg.ret < 0)
4800023     {
4800024         //
4800025         // Something wrong.
4800026         //
4800027         errno = msg.errno;
4800028         errln = msg.errln;
4800029         strncpy (errfn, msg.errfn, PATH_MAX);
4800030     }
4800031     //
4800032     // Return.
4800033     //
4800034     return (msg.ret);
```

4800035

}

95.21.7 lib/sys/os32/sys.s

Si veda la sezione [87.56](#).

```

4810001  .global sys
4810002  #-----
4810003  .text
4810004  #-----
4810005  # Call a system call.
4810006  #
4810007  # Please remember that system calls should never be
4810008  # used (called) inside the kernel code, because system
4810009  # calls cannot be nested for the os32 simple
4810010  # architecture!
4810011  # If a particular function is necessary inside the
4810012  # kernel, that usually is made by a system call, an
4810013  # appropriate k_...() function must be made, to avoid
4810014  # the problem.
4810015  #
4810016  #-----
4810017  .align 4
4810018  sys:
4810019      int    $128    # 0x80
4810020      ret

```

95.21.8 lib/sys/os32/umount.c

Si veda la sezione [87.36](#).

```

4820001  #include <sys/types.h>
4820002  #include <errno.h>
4820003  #include <sys/os32.h>
4820004  #include <stddef.h>
4820005  #include <string.h>
4820006  //-----

```

```
4820007 int
4820008 umount (const char *path_mnt)
4820009 {
4820010     sysmsg_umount_t msg;
4820011     //
4820012     msg.path_mnt = path_mnt;
4820013     msg.ret = 0;
4820014     msg.errno = 0;
4820015     //
4820016     sys (SYS_UMOUNT, &msg, (sizeof msg));
4820017     //
4820018     errno = msg.errno;
4820019     errln = msg.errln;
4820020     strncpy (errfn, msg.errfn, PATH_MAX);
4820021     return (msg.ret);
4820022 }
```

95.21.9 lib/sys/os32/z_perror.c

<<

Si veda la sezione [87.65](#).

```
4830001 #include <sys/os32.h>
4830002 #include <errno.h>
4830003 #include <stddef.h>
4830004 #include <string.h>
4830005 //-----
4830006 void
4830007 z_perror (const char *string)
4830008 {
4830009     //
4830010     // If errno is zero, there is nothing to show.
4830011     //
4830012     if (errno == 0)
4830013     {
4830014         return;
4830015     }
4830016     //
```

```
4830017 // Show the string if there is one.
4830018 //
4830019 if (string != NULL && strlen (string) > 0)
4830020 {
4830021     z_printf ("%s: ", string);
4830022 }
4830023 //
4830024 // Show the translated error.
4830025 //
4830026 if (errfn[0] != 0 && errln != 0)
4830027 {
4830028     z_printf ("%s:%u:%i] %s\n",
4830029             errfn, errln, errno, strerror (errno));
4830030 }
4830031 else
4830032 {
4830033     z_printf ("%i] %s\n", errno, strerror (errno));
4830034 }
4830035 }
```

95.21.10 lib/sys/os32/z_printf.c

Si veda la sezione [87.65](#).

```
4840001 #include <sys/os32.h>
4840002 #include <restrict.h>
4840003 //-----
4840004 int
4840005 z_printf (const char *restrict format, ...)
4840006 {
4840007     va_list ap;
4840008     va_start (ap, format);
4840009     return z_vprintf (format, ap);
4840010 }
```



95.21.11 lib/sys/os32/z_vprintf.c

«

Si veda la sezione [87.65](#).

```
4850001 #include <sys/os32.h>
4850002 #include <restrict.h>
4850003 //-----
4850004 int
4850005 z_vprintf (const char *restrict format, va_list arg)
4850006 {
4850007     int ret;
4850008     sysmsg_zpstring_t msg;
4850009     msg.string[0] = 0;
4850010     ret = vsprintf (msg.string, format, arg);
4850011     sys (SYS_ZPSTRING, &msg, (sizeof msg));
4850012     return ret;
4850013 }
```

95.22 os32: «lib/sys/sa_family_t.h»

«

Si veda la sezione [91.3](#).

```
4860001 #ifndef _SYS_SA_FAMILY_T_H
4860002 #define _SYS_SA_FAMILY_T_H    1
4860003 //-----
4860004 #include <inttypes.h>
4860005 //-----
4860006 typedef uint16_t sa_family_t;    // Address family.
4860007 //-----
4860008 #endif
```


95.23 os32: «lib/sys/socket.h»



Si veda la sezione [91.3](#).

```
4870001 #ifndef _SYS_SOCKET_H
4870002 #define _SYS_SOCKET_H      1
4870003 //-----
4870004 #include <stdint.h>
4870005 #include <unistd.h>
4870006 #include <sys/socklen_t.h>
4870007 #include <sys/sa_family_t.h>
4870008 //-----
4870009 struct sockaddr
4870010 {
4870011     sa_family_t sa_family;          // Address family.
4870012     char sa_data[14];              // Socket address.
4870013 };
4870014 //
4870015 //
4870016 //
4870017 struct sockaddr_storage
4870018 {
4870019     sa_family_t ss_family;          // Socket storage
4870020     // family.
4870021     uint8_t ss_zero[14];           // Filler.
4870022 };
4870023 //
4870024 //
4870025 //
4870026 #define SOCK_STREAM      1          // Byte-stream socket.
4870027 #define SOCK_DGRAM      2          // Datagram socket.
4870028 #define SOCK_RAW        3          // Raw protocol
4870029                                // interface.
4870030 #define SOCK_SEQPACKET  5          // Sequenced-packet
4870031                                // socket.
4870032 //
4870033 // Protocol families:
4870034 //
```

```

4870035 #define PF_UNSPEC      0          // Unspecified.
4870036 #define PF_UNIX       1          // Unix domain socket.
4870037 #define PF_INET       2          // IPv4 protocol
4870038                                     // family.
4870039 #define PF_INET6      10         // IPv6 protocol
4870040                                     // family.
4870041 //
4870042 // Address families.
4870043 //
4870044 #define AF_UNSPEC     PF_UNSPEC   // Unspecified.
4870045 #define AF_UNIX      PF_UNIX     // Unix domain socket.
4870046 #define AF_INET      PF_INET     // IPv4 address
4870047                                     // family.
4870048 #define AF_INET6     PF_INET6    // IPv6 address
4870049                                     // family.
4870050 //-----
4870051 int accept (int sfdn, struct sockaddr *addr,
4870052             socklen_t * addrlen);
4870053 int bind (int sfdn, const struct sockaddr *addr,
4870054           socklen_t addrlen);
4870055 int connect (int sfdn, const struct sockaddr *addr,
4870056             socklen_t addrlen);
4870057 int listen (int sfdn, int backlog);
4870058 ssize_t send (int sfdn, const void *buffer,
4870059             size_t count, int flags);
4870060 ssize_t recvfrom (int sfdn, void *buffer, size_t count,
4870061                 int flags, struct sockaddr *addrfrom,
4870062                 socklen_t * addrlen);
4870063 int socket (int family, int type, int protocol);
4870064
4870065 #define recv(sfdn, buffer, count, flags) \
4870066     recvfrom (sfdn, buffer, count, flags, NULL, NULL)
4870067 //-----
4870068 #endif

```

Sorgenti della libreria generale	2131
95.23.2 lib/sys/socket/bind.c	2133
95.23.3 lib/sys/socket/connect.c	2134
95.23.4 lib/sys/socket/listen.c	2136
95.23.5 lib/sys/socket/recvfrom.c	2137
95.23.6 lib/sys/socket/send.c	2140
95.23.7 lib/sys/socket/socket.c	2142

95.23.1 lib/sys/socket/accept.c



Si veda la sezione [87.3](#).

```
4880001 #include <sys/os32.h>
4880002 #include <errno.h>
4880003 #include <string.h>
4880004 #include <stdio.h>
4880005 #include <fcntl.h>
4880006 //-----
4880007 int
4880008 accept (int sfdn, struct sockaddr *addr,
4880009         socklen_t * addrlen)
4880010 {
4880011     sysmsg_accept_t msg;
4880012     //
4880013     // Fill the message.
4880014     //
4880015     msg.sfdn = sfdn;
4880016     memset (&msg.addr, 0x00, sizeof (msg.addr));
4880017     msg.addrlen = *addrlen;
4880018     msg.fl_flags = 0;      // Not necessary.
4880019     msg.ret = 0;
4880020     //
4880021     // Syscall.
4880022     //
4880023     while (1)
```

```
4880024     {
4880025         sys (SYS_ACCEPT, &msg, (sizeof msg));
4880026         //
4880027         if (msg.ret < 0
4880028             && (msg.errno == EAGAIN
4880029                 || msg.errno == EWOULDBLOCK))
4880030         {
4880031             //
4880032             // No request at the moment.
4880033             //
4880034             if (msg.fl_flags & O_NONBLOCK)
4880035             {
4880036                 //
4880037                 // Don't block.
4880038                 //
4880039                 break;
4880040             }
4880041             else
4880042             {
4880043                 //
4880044                 // Keep trying.
4880045                 //
4880046                 continue;
4880047             }
4880048         }
4880049         else
4880050         {
4880051             break;
4880052         }
4880053     }
4880054     //
4880055     // Check return value.
4880056     //
4880057     if (msg.ret < 0)
4880058     {
4880059         //
4880060         // Something wrong.
```

```
4880061     //
4880062     errno = msg.errno;
4880063     errln = msg.errln;
4880064     strncpy (errfn, msg.errfn, PATH_MAX);
4880065 }
4880066 else
4880067 {
4880068     //
4880069     // Update the socket address and the address
4880070     // length.
4880071     //
4880072     if (addrlen != NULL && addr != NULL && *addrlen > 0)
4880073     {
4880074         memcpy (addr, &msg.addr,
4880075             min (msg.addrlen, *addrlen));
4880076         *addrlen = msg.addrlen;
4880077     }
4880078 }
4880079 //
4880080 // Return.
4880081 //
4880082 return (msg.ret);
4880083 }
```

95.23.2 lib/sys/socket/bind.c

Si veda la sezione [87.4](#).

```
4890001 #include <sys/os32.h>
4890002 #include <errno.h>
4890003 #include <string.h>
4890004 #include <stdio.h>
4890005 //-----
4890006 int
4890007 bind (int sfdn, const struct sockaddr *addr,
4890008     socklen_t addrlen)
4890009 {
```



```
4890010 sysmsg_bind_t msg;
4890011 //
4890012 // Fill the message.
4890013 //
4890014 msg.sfdn = sfdn;
4890015 memcpy (&msg.addr, addr, (size_t) addrlen);
4890016 msg.addrlen = addrlen;
4890017 msg.ret = 0;
4890018 //
4890019 // Syscall.
4890020 //
4890021 sys (SYS_BIND, &msg, (sizeof msg));
4890022 //
4890023 // Check return value.
4890024 //
4890025 if (msg.ret < 0)
4890026 {
4890027     //
4890028     // Something wrong.
4890029     //
4890030     errno = msg.errno;
4890031     errln = msg.errln;
4890032     strncpy (errfn, msg.errfn, PATH_MAX);
4890033 }
4890034 //
4890035 // Return.
4890036 //
4890037 return (msg.ret);
4890038 }
```

95.23.3 lib/sys/socket/connect.c



Si veda la sezione [87.11](#).

```
4900001 #include <sys/os32.h>
4900002 #include <errno.h>
4900003 #include <string.h>
```

```
4900004 #include <stdio.h>
4900005 //-----
4900006 int
4900007 connect (int sfdn, const struct sockaddr *addr,
4900008         socklen_t addrlen)
4900009 {
4900010     sysmsg_connect_t msg;
4900011     //
4900012     // Fill the message.
4900013     //
4900014     msg.sfdn = sfdn;
4900015     memcpy (&msg.addr, addr, (size_t) addrlen);
4900016     msg.addrlen = addrlen;
4900017     msg.ret = 0;
4900018     //
4900019     // Syscall.
4900020     //
4900021     while (1)
4900022     {
4900023         sys (SYS_CONNECT, &msg, (sizeof msg));
4900024         //
4900025         if (msg.ret < 0)
4900026         {
4900027             if (msg.errno == EINPROGRESS
4900028                 || msg.errno == EALREADY)
4900029             {
4900030                 //
4900031                 // Loop until the connection is
4900032                 // established, or a
4900033                 // different error comes.
4900034                 //
4900035                 continue;
4900036             }
4900037             else
4900038             {
4900039                 break;
4900040             }
4900040         }
```

```
4900041     }
4900042     else
4900043     {
4900044         break;
4900045     }
4900046 }
4900047 //
4900048 // Check return value.
4900049 //
4900050 if (msg.ret < 0)
4900051 {
4900052     //
4900053     // Something wrong.
4900054     //
4900055     errno = msg.errno;
4900056     errln = msg.errln;
4900057     strncpy (errfn, msg.errfn, PATH_MAX);
4900058 }
4900059 //
4900060 // Return.
4900061 //
4900062 return (msg.ret);
4900063 }
```

95.23.4 lib/sys/socket/listen.c

«

Si veda la sezione [87.31](#).

```
4910001 #include <sys/os32.h>
4910002 #include <errno.h>
4910003 #include <string.h>
4910004 #include <stdio.h>
4910005 //-----
4910006 int
4910007 listen (int sfdn, int backlog)
4910008 {
4910009     sysmsg_listen_t msg;
```



```
4910010 //
4910011 // Fill the message.
4910012 //
4910013 msg.sfdn = sfdn;
4910014 msg.backlog = backlog;
4910015 msg.ret = 0;
4910016 //
4910017 // Syscall.
4910018 //
4910019 sys (SYS_LISTEN, &msg, (sizeof msg));
4910020 //
4910021 // Check return value.
4910022 //
4910023 if (msg.ret < 0)
4910024 {
4910025     //
4910026     // Something wrong.
4910027     //
4910028     errno = msg.errno;
4910029     errln = msg.errln;
4910030     strncpy (errfn, msg.errfn, PATH_MAX);
4910031 }
4910032 //
4910033 // Return.
4910034 //
4910035 return (msg.ret);
4910036 }
```

95.23.5 lib/sys/socket/recvfrom.c

Si veda la sezione [87.40](#).

```
4920001 #include <sys/os32.h>
4920002 #include <errno.h>
4920003 #include <string.h>
4920004 #include <stdio.h>
4920005 #include <fcntl.h>
```



```
4920006 //-----
4920007 ssize_t
4920008 recvfrom (int sfdn, void *buffer, size_t count,
4920009           int flags, struct sockaddr *addrfrom,
4920010           socklen_t * addrlen)
4920011 {
4920012     sysmsg_recvfrom_t msg;
4920013     //
4920014     // Reduce size of read if necessary.
4920015     //
4920016     if (count > BUFSIZ)
4920017     {
4920018         count = BUFSIZ;
4920019     }
4920020     //
4920021     // Fill the message.
4920022     //
4920023     msg.sfdn = sfdn;
4920024     msg.buffer = buffer;
4920025     msg.count = count;
4920026     msg.flags = flags;
4920027     msg.addrfrom = addrfrom;
4920028     msg.addrsz = addrlen;
4920029     msg.fl_flags = 0;      // Not necessary.
4920030     msg.ret = 0;
4920031     //
4920032     // Repeat syscall, until something is received or
4920033     // end of file is
4920034     // reached.
4920035     //
4920036     while (1)
4920037     {
4920038         sys (SYS_RECVFROM, &msg, (sizeof msg));
4920039         if (msg.ret == 0)
4920040         {
4920041             //
4920042             // Stream closed from the other side.
```

```
4920043         //
4920044         break;
4920045     }
4920046     if (msg.ret < 0
4920047         && (msg.errno == EAGAIN
4920048             || msg.errno == EWOULDBLOCK))
4920049     {
4920050         //
4920051         // No data at the moment.
4920052         //
4920053         if (msg.fl_flags & O_NONBLOCK)
4920054         {
4920055             //
4920056             // Don't block.
4920057             //
4920058             break;
4920059         }
4920060         else
4920061         {
4920062             //
4920063             // Keep trying.
4920064             //
4920065             continue;
4920066         }
4920067     }
4920068     //
4920069     // Otherwise, we have received something.
4920070     //
4920071     break;
4920072 }
4920073 //
4920074 //
4920075 //
4920076 if (msg.ret < 0)
4920077 {
4920078     //
4920079     // No valid read.
```

```

4920080      //
4920081      errno = msg.errno;
4920082      errln = msg.errln;
4920083      strncpy (errfn, msg.errfn, PATH_MAX);
4920084      return (msg.ret);
4920085  }
4920086  //
4920087  if (msg.ret > count)
4920088  {
4920089      //
4920090      // A strange value was returned. Considering it
4920091      // a read error.
4920092      //
4920093      errset (EIO);      // I/O error.
4920094      return (-1);
4920095  }
4920096  //
4920097  // A valid read: return.
4920098  //
4920099  return (msg.ret);
4920100  }

```

95.23.6 lib/sys/socket/send.c

«

Si veda la sezione [87.45](#).

```

4930001  #include <unistd.h>
4930002  #include <sys/os32.h>
4930003  #include <errno.h>
4930004  #include <string.h>
4930005  #include <stdio.h>
4930006  //-----
4930007  ssize_t
4930008  send (int sfdn, const void *buffer, size_t count, int flags)
4930009  {
4930010      sysmsg_send_t msg;
4930011      int retry = 3;

```

```
4930012 //
4930013 // Reduce size of write if necessary.
4930014 //
4930015 if (count > BUFSIZ)
4930016     {
4930017     count = BUFSIZ;
4930018     }
4930019 //
4930020 // Fill the message.
4930021 //
4930022 msg.sfdn = sfdn;
4930023 msg.buffer = buffer;
4930024 msg.count = count;
4930025 msg.flags = flags;
4930026 //
4930027 // Syscall.
4930028 //
4930029 for (; retry > 0; retry--)
4930030     {
4930031     sys (SYS_SEND, &msg, (sizeof msg));
4930032     //
4930033     // Check.
4930034     //
4930035     if ((msg.ret < 0) && (msg.errno == E_ARP_MISSING))
4930036         {
4930037         sleep (1);
4930038         continue; // Retry.
4930039         }
4930040     else
4930041         {
4930042         break;
4930043         }
4930044     }
4930045 //
4930046 // Check the final result and return.
4930047 //
4930048 if (msg.ret < 0)
```

```
4930049     {
4930050         //
4930051         // No valid write.
4930052         //
4930053         errno = msg.errno;
4930054         errln = msg.errln;
4930055         strncpy (errfn, msg.errfn, PATH_MAX);
4930056         return (msg.ret);
4930057     }
4930058     //
4930059     if (msg.ret > count)
4930060     {
4930061         //
4930062         // A strange value was returned. Considering it
4930063         // a read error.
4930064         //
4930065         errset (EIO);      // I/O error.
4930066         return (-1);
4930067     }
4930068     //
4930069     // A valid write return.
4930070     //
4930071     return (msg.ret);
4930072 }
```

95.23.7 lib/sys/socket/socket.c

«

Si veda la sezione [87.54](#).

```
4940001 #include <sys/os32.h>
4940002 #include <errno.h>
4940003 #include <string.h>
4940004 #include <stdio.h>
4940005 //-----
4940006 int
4940007 socket (int family, int type, int protocol)
4940008 {
```

```

4940009     sysmsg_socket_t msg;
4940010     //
4940011     // Fill the message.
4940012     //
4940013     msg.family = family;
4940014     msg.type = type;
4940015     msg.protocol = protocol;
4940016     msg.ret = 0;
4940017     //
4940018     // Syscall.
4940019     //
4940020     sys (SYS_SOCKET, &msg, (sizeof msg));
4940021     //
4940022     // Check return value.
4940023     //
4940024     if (msg.ret < 0)
4940025     {
4940026         //
4940027         // Something wrong.
4940028         //
4940029         errno = msg.errno;
4940030         errln = msg.errln;
4940031         strncpy (errfn, msg.errfn, PATH_MAX);
4940032     }
4940033     //
4940034     // Return.
4940035     //
4940036     return (msg.ret);
4940037 }

```

95.24 os32: «lib/sys/socklen_t.h»

Si veda la sezione [91.3](#).

```

4950001 #ifndef _SYS_SOCKLEN_T_H
4950002 #define _SYS_SOCKLEN_T_H      1
4950003 //-----

```

```
4950004 #include <stdint.h>
4950005 //-----
4950006 typedef uint32_t socklen_t;
4950007 //-----
4950008 #endif
```

95.25 os32: «lib/sys/stat.h»

«

Si veda la sezione [91.3](#).

```
4960001 #ifndef _SYS_STAT_H
4960002 #define _SYS_STAT_H      1
4960003
4960004 #include <restrict.h>
4960005 #include <sys/types.h> // dev_t
4960006                        // off_t
4960007                        // blkcnt_t
4960008                        // blksize_t
4960009                        // ino_t
4960010                        // mode_t
4960011                        // nlink_t
4960012                        // uid_t
4960013                        // gid_t
4960014                        // time_t
4960015 //-----
4960016 // File type.
4960017 //-----
4960018 #define S_IFMT      0170000 // File type mask.
4960019 //
4960020 #define S_IFBLK    0060000 // Block device file.
4960021 #define S_IFCHR    0020000 // Character device
4960022                        // file.
4960023 #define S_IFIFO    0010000 // Pipe (FIFO) file.
4960024 #define S_IFREG    0100000 // Regular file.
4960025 #define S_IFDIR    0040000 // Directory.
```



```
4960026 #define S_IFLNK 0120000 // Symbolic link.
4960027 #define S_IFSOCK 0140000 // Unix domain socket.
4960028 //-----
4960029 // Owner user access permissions.
4960030 //-----
4960031 #define S_IRWXU 0000700 // Owner user access
4960032 // permissions mask.
4960033 //
4960034 #define S_IRUSR 0000400 // Owner user read
4960035 // access permission.
4960036 #define S_IWUSR 0000200 // Owner user write
4960037 // access permission.
4960038 #define S_IXUSR 0000100 // Owner user
4960039 // execution or cross
4960040 // perm.
4960041 //-----
4960042 // Group owner access permissions.
4960043 //-----
4960044 #define S_IRWXG 0000070 // Owner group access
4960045 // permissions mask.
4960046 //
4960047 #define S_IRGRP 0000040 // Owner group read
4960048 // access permission.
4960049 #define S_IWGRP 0000020 // Owner group write
4960050 // access permission.
4960051 #define S_IXGRP 0000010 // Owner group
4960052 // execution or cross
4960053 // perm.
4960054 //-----
4960055 // Other users access permissions.
4960056 //-----
4960057 #define S_IRWXO 0000007 // Other users access
4960058 // permissions mask.
4960059 //
4960060 #define S_IROTH 0000004 // Other users read
4960061 // access permission.
4960062 #define S_IWOTH 0000002 // Other users write
```

```
4960063                                     // access permissions.
4960064 #define S_IXOTH  0000001                // Other users
4960065                                     // execution or cross
4960066                                     // perm.
4960067 //-----
4960068 // S-bit: in this case there is no mask to select all
4960069 // of them.
4960070 //-----
4960071 #define S_ISUID   0004000                // S-UID.
4960072 #define S_ISGID   0002000                // S-GID.
4960073 #define S_ISVTX   0001000                // Sticky.
4960074 //-----
4960075 // Macroinstructions to verify the type of file.
4960076 //-----
4960077 //
4960078 // Block device:
4960079 //
4960080 #define S_ISBLK(m)  ((m) & S_IFMT) == S_IFBLK)
4960081 //
4960082 // Character device:
4960083 //
4960084 #define S_ISCHR(m)  ((m) & S_IFMT) == S_IFCHR)
4960085 //
4960086 // FIFO.
4960087 //
4960088 #define S_ISFIFO(m) ((m) & S_IFMT) == S_IFIFO)
4960089 //
4960090 // Regular file.
4960091 //
4960092 #define S_ISREG(m)  ((m) & S_IFMT) == S_IFREG)
4960093 //
4960094 // Directory.
4960095 //
4960096 #define S_ISDIR(m)  ((m) & S_IFMT) == S_IFDIR)
4960097 //
4960098 // Symbolic link.
4960099 //
```

```
4960100 #define S_ISLNK(m)      (((m) & S_IFMT) == S_IFLNK)
4960101 //
4960102 // Socket (Unix domain socket).
4960103 //
4960104 #define S_ISSOCK(m)    (((m) & S_IFMT) == S_IFSOCK)
4960105 //-----
4960106 // Structure 'stat'.
4960107 //-----
4960108 struct stat
4960109 {
4960110     dev_t st_dev; // Device containing the file.
4960111     ino_t st_ino; // File serial number (inode number).
4960112     mode_t st_mode; // File type and permissions.
4960113     nlink_t st_nlink; // Links to the file.
4960114     uid_t st_uid; // Owner user id.
4960115     gid_t st_gid; // Owner group id.
4960116     dev_t st_rdev; // Device number if it is a
4960117 // device file.
4960118     off_t st_size; // File size.
4960119     time_t st_atime; // Last access time.
4960120     time_t st_mtime; // Last modification time.
4960121     time_t st_ctime; // Last inode modification.
4960122     blksize_t st_blksize; // Block size for I/O
4960123 // operations.
4960124     blkcnt_t st_blocks; // File size / block size.
4960125 };
4960126 //-----
4960127 // Function prototypes.
4960128 //-----
4960129 int chmod (const char *path, mode_t mode);
4960130 int fchmod (int fdn, mode_t mode);
4960131 int fstat (int fdn, struct stat *buffer);
4960132 int lstat (const char *restrict path,
4960133           struct stat *restrict buffer);
4960134 int mkdir (const char *path, mode_t mode);
4960135 int mkfifo (const char *path, mode_t mode);
4960136 int mknod (const char *path, mode_t mode, dev_t dev);
```

```

4960137 int stat (const char *restrict path,
4960138         struct stat *restrict buffer);
4960139 mode_t umask (mode_t mask);
4960140
4960141 #endif // _SYS_STAT_H

```

95.25.1	lib/sys/stat/chmod.c	2148
95.25.2	lib/sys/stat/fchmod.c	2149
95.25.3	lib/sys/stat/fstat.c	2150
95.25.4	lib/sys/stat/mkdir.c	2151
95.25.5	lib/sys/stat/mknod.c	2152
95.25.6	lib/sys/stat/stat.c	2152
95.25.7	lib/sys/stat/umask.c	2154

95.25.1 lib/sys/stat/chmod.c



Si veda la sezione [87.7](#).

```

4970001 #include <sys/stat.h>
4970002 #include <string.h>
4970003 #include <sys/os32.h>
4970004 #include <errno.h>
4970005 #include <limits.h>
4970006 //-----
4970007 int
4970008 chmod (const char *path, mode_t mode)
4970009 {
4970010     sysmsg_chmod_t msg;
4970011     //
4970012     msg.path = path;
4970013     msg.mode = mode;
4970014     //

```

```
4970015     sys (SYS_CHMOD, &msg, (sizeof msg));
4970016     //
4970017     errno = msg.errno;
4970018     errln = msg.errln;
4970019     strncpy (errfn, msg.errfn, PATH_MAX);
4970020     return (msg.ret);
4970021 }
```

95.25.2 lib/sys/stat/fchmod.c

Si veda la sezione [87.7](#).

```
4980001 #include <sys/stat.h>
4980002 #include <string.h>
4980003 #include <sys/os32.h>
4980004 #include <errno.h>
4980005 #include <limits.h>
4980006 //-----
4980007 int
4980008 fchmod (int fdn, mode_t mode)
4980009 {
4980010     sysmsg_fchmod_t msg;
4980011     //
4980012     msg.fdn = fdn;
4980013     msg.mode = mode;
4980014     //
4980015     sys (SYS_FCHMOD, &msg, (sizeof msg));
4980016     //
4980017     errno = msg.errno;
4980018     errln = msg.errln;
4980019     strncpy (errfn, msg.errfn, PATH_MAX);
4980020     return (msg.ret);
4980021 }
```

95.25.3 lib/sys/stat/fstat.c



Si veda la sezione [87.55](#).

```
4990001 #include <unistd.h>
4990002 #include <errno.h>
4990003 #include <sys/os32.h>
4990004 #include <string.h>
4990005 //-----
4990006 int
4990007 fstat (int fdn, struct stat *buffer)
4990008 {
4990009     sysmsg_fstat_t msg;
4990010     //
4990011     msg.fdn = fdn;
4990012     msg.stat.st_dev = buffer->st_dev;
4990013     msg.stat.st_ino = buffer->st_ino;
4990014     msg.stat.st_mode = buffer->st_mode;
4990015     msg.stat.st_nlink = buffer->st_nlink;
4990016     msg.stat.st_uid = buffer->st_uid;
4990017     msg.stat.st_gid = buffer->st_gid;
4990018     msg.stat.st_rdev = buffer->st_rdev;
4990019     msg.stat.st_size = buffer->st_size;
4990020     msg.stat.st_atime = buffer->st_atime;
4990021     msg.stat.st_mtime = buffer->st_mtime;
4990022     msg.stat.st_ctime = buffer->st_ctime;
4990023     msg.stat.st_blksize = buffer->st_blksize;
4990024     msg.stat.st_blocks = buffer->st_blocks;
4990025     //
4990026     sys (SYS_FSTAT, &msg, (sizeof msg));
4990027     //
4990028     buffer->st_dev = msg.stat.st_dev;
4990029     buffer->st_ino = msg.stat.st_ino;
4990030     buffer->st_mode = msg.stat.st_mode;
4990031     buffer->st_nlink = msg.stat.st_nlink;
4990032     buffer->st_uid = msg.stat.st_uid;
4990033     buffer->st_gid = msg.stat.st_gid;
4990034     buffer->st_rdev = msg.stat.st_rdev;
```

```
4990035     buffer->st_size = msg.stat.st_size;
4990036     buffer->st_atime = msg.stat.st_atime;
4990037     buffer->st_mtime = msg.stat.st_mtime;
4990038     buffer->st_ctime = msg.stat.st_ctime;
4990039     buffer->st_blksize = msg.stat.st_blksize;
4990040     buffer->st_blocks = msg.stat.st_blocks;
4990041     //
4990042     errno = msg.errno;
4990043     errln = msg.errln;
4990044     strncpy (errfn, msg.errfn, PATH_MAX);
4990045     return (msg.ret);
4990046 }
```

95.25.4 lib/sys/stat/mkdir.c

Si veda la sezione [87.34](#).



```
5000001 #include <sys/stat.h>
5000002 #include <string.h>
5000003 #include <sys/os32.h>
5000004 #include <errno.h>
5000005 #include <limits.h>
5000006 //-----
5000007 int
5000008 mkdir (const char *path, mode_t mode)
5000009 {
5000010     sysmsg_mkdir_t msg;
5000011     //
5000012     msg.path = path;
5000013     msg.mode = mode;
5000014     //
5000015     sys (SYS_MKDIR, &msg, (sizeof msg));
5000016     //
5000017     errno = msg.errno;
5000018     errln = msg.errln;
5000019     strncpy (errfn, msg.errfn, PATH_MAX);
5000020     return (msg.ret);
```

5000021	}
---------	---

95.25.5 lib/sys/stat/mknod.c

<<

Si veda la sezione [87.35](#).

```
5010001 #include <unistd.h>
5010002 #include <errno.h>
5010003 #include <sys/os32.h>
5010004 #include <string.h>
5010005 //-----
5010006 int
5010007 mknod (const char *path, mode_t mode, dev_t device)
5010008 {
5010009     sysmsg_mknod_t msg;
5010010     //
5010011     msg.path = path;
5010012     msg.mode = mode;
5010013     msg.device = device;
5010014     //
5010015     sys (SYS_MKNOD, &msg, (sizeof msg));
5010016     //
5010017     errno = msg.errno;
5010018     errln = msg.errln;
5010019     strncpy (errfn, msg.errfn, PATH_MAX);
5010020     return (msg.ret);
5010021 }
```

95.25.6 lib/sys/stat/stat.c

<<

Si veda la sezione [87.55](#).

```
5020001 #include <unistd.h>
5020002 #include <errno.h>
5020003 #include <sys/os32.h>
5020004 #include <string.h>
5020005 //-----
```



```
5020006 int
5020007 stat (const char *path, struct stat *buffer)
5020008 {
5020009     sysmsg_stat_t msg;
5020010     //
5020011     msg.path = path;
5020012     //
5020013     msg.stat.st_dev = buffer->st_dev;
5020014     msg.stat.st_ino = buffer->st_ino;
5020015     msg.stat.st_mode = buffer->st_mode;
5020016     msg.stat.st_nlink = buffer->st_nlink;
5020017     msg.stat.st_uid = buffer->st_uid;
5020018     msg.stat.st_gid = buffer->st_gid;
5020019     msg.stat.st_rdev = buffer->st_rdev;
5020020     msg.stat.st_size = buffer->st_size;
5020021     msg.stat.st_atime = buffer->st_atime;
5020022     msg.stat.st_mtime = buffer->st_mtime;
5020023     msg.stat.st_ctime = buffer->st_ctime;
5020024     msg.stat.st_blksize = buffer->st_blksize;
5020025     msg.stat.st_blocks = buffer->st_blocks;
5020026     //
5020027     sys (SYS_STAT, &msg, (sizeof msg));
5020028     //
5020029     buffer->st_dev = msg.stat.st_dev;
5020030     buffer->st_ino = msg.stat.st_ino;
5020031     buffer->st_mode = msg.stat.st_mode;
5020032     buffer->st_nlink = msg.stat.st_nlink;
5020033     buffer->st_uid = msg.stat.st_uid;
5020034     buffer->st_gid = msg.stat.st_gid;
5020035     buffer->st_rdev = msg.stat.st_rdev;
5020036     buffer->st_size = msg.stat.st_size;
5020037     buffer->st_atime = msg.stat.st_atime;
5020038     buffer->st_mtime = msg.stat.st_mtime;
5020039     buffer->st_ctime = msg.stat.st_ctime;
5020040     buffer->st_blksize = msg.stat.st_blksize;
5020041     buffer->st_blocks = msg.stat.st_blocks;
5020042     //
```

```
5020043     errno = msg.errno;
5020044     errln = msg.errln;
5020045     strncpy (errfn, msg.errfn, PATH_MAX);
5020046     return (msg.ret);
5020047 }
```

95.25.7 lib/sys/stat/umask.c

«

Si veda la sezione [87.60](#).

```
5030001 #include <sys/stat.h>
5030002 #include <string.h>
5030003 #include <sys/os32.h>
5030004 #include <errno.h>
5030005 #include <limits.h>
5030006 //-----
5030007 mode_t
5030008 umask (mode_t mask)
5030009 {
5030010     sysmsg_umask_t msg;
5030011     msg.umask = mask;
5030012     sys (SYS_UMASK, &msg, (sizeof msg));
5030013     return (msg.ret);
5030014 }
```

95.26 os32: «lib/sys/types.h»

«

Si veda la sezione [91.3](#).

```
5040001 #ifndef _SYS_TYPES_H
5040002 #define _SYS_TYPES_H    1
5040003 //-----
5040004 #include <clock_t.h>
5040005 #include <time_t.h>
5040006 #include <size_t.h>
5040007 //-----
```

```

5040008 typedef long int blkcnt_t;
5040009 typedef long int blksize_t;
5040010 typedef uint16_t dev_t; // Traditional device size.
5040011 typedef unsigned int id_t;
5040012 typedef unsigned int gid_t;
5040013 typedef unsigned int uid_t;
5040014 typedef uint16_t ino_t; // Minix 1 file system inode
5040015                       // size.
5040016 typedef uint16_t mode_t; // Minix 1 file system
5040017                       // mode size.
5040018 typedef unsigned int nlink_t;
5040019 typedef long long int off_t;
5040020 typedef int pid_t;
5040021 typedef unsigned long int pthread_t;
5040022 typedef int ssize_t;
5040023 //-----
5040024 // Common extentions.
5040025 //
5040026 dev_t makedev (int major, int minor);
5040027 int major (dev_t device);
5040028 int minor (dev_t device);
5040029 //-----
5040030 #endif

```

[95.26.1](#) [lib/sys/types/major.c](#) [2155](#)

[95.26.2](#) [lib/sys/types/makedev.c](#) [2156](#)

[95.26.3](#) [lib/sys/types/minor.c](#) [2156](#)

[95.26.1](#) [lib/sys/types/major.c](#)

Si veda la sezione [88.75](#).

```

5050001 #include <sys/types.h>
5050002 //-----
5050003 int

```

```
5050004 major (dev_t device)
5050005 {
5050006     return ((int) (device / 256));
5050007 }
```

95.26.2 lib/sys/types/makedev.c

<<

Si veda la sezione [88.75](#).

```
5060001 #include <sys/types.h>
5060002 //-----
5060003 dev_t
5060004 makedev (int major, int minor)
5060005 {
5060006     return ((dev_t) (major * 256 + minor));
5060007 }
```

95.26.3 lib/sys/types/minor.c

<<

Si veda la sezione [88.75](#).

```
5070001 #include <sys/types.h>
5070002 //-----
5070003 int
5070004 minor (dev_t device)
5070005 {
5070006     return ((dev_t) (device & 0x00FF));
5070007 }
```

95.27 os32: «lib/sys/wait.h»

<<

Si veda la sezione [91.3](#).

```
5080001 #ifndef _SYS_WAIT_H
5080002 #define _SYS_WAIT_H      1
5080003
```

```

5080004 #include <sys/types.h>
5080005
5080006 //-----
5080007 pid_t wait (int *status);
5080008 //-----
5080009
5080010 #endif

```

95.27.1 lib/sys/wait/wait.c 2157

95.27.1 lib/sys/wait/wait.c

Si veda la sezione [87.63](#).



```

5090001 #include <sys/types.h>
5090002 #include <errno.h>
5090003 #include <sys/os32.h>
5090004 #include <stddef.h>
5090005 #include <string.h>
5090006 //-----
5090007 pid_t
5090008 wait (int *status)
5090009 {
5090010     sysmsg_wait_t msg;
5090011     msg.ret = 0;
5090012     msg.errno = 0;
5090013     msg.status = 0;
5090014     while (msg.ret == 0)
5090015     {
5090016         //
5090017         // Loop as long as there are children, an none
5090018         // is dead.
5090019         //
5090020         sys (SYS_WAIT, &msg, (sizeof msg));
5090021     }
5090022     errno = msg.errno;
5090023     errln = msg.errln;

```

```

5090024     strncpy (errfn, msg.errfn, PATH_MAX);
5090025     //
5090026     if (status != NULL)
5090027     {
5090028         //
5090029         // Only the low eight bits are returned.
5090030         //
5090031         *status = (msg.status & 0x00FF);
5090032     }
5090033     return (msg.ret);
5090034 }

```

95.28 os32: «lib/termios.h»

«

Si veda la sezione 87.58.

```

5100001 #ifndef _TERMIOS_H
5100002 #define _TERMIOS_H      1
5100003 //-----
5100004 #include <stdint.h>
5100005 //-----
5100006 typedef uint16_t tcflag_t;
5100007 typedef unsigned char cc_t;
5100008 //-----
5100009 #define NCCS      11      // 'c_cc[]' size.
5100010 //
5100011 struct termios
5100012 {
5100013     tcflag_t c_iflag;
5100014     tcflag_t c_oflag;
5100015     tcflag_t c_cflag;
5100016     tcflag_t c_lflag;
5100017     cc_t c_cc[NCCS];
5100018 };
5100019 //
5100020 // Subscript names for 'c_cc[]' array, inside the
5100021 // 'termios' structure:

```

```
5100022 //
5100023 #define VEOF      0      // EOF character
5100024 #define VEOL     1      // EOL character
5100025 #define VERASE   2      // ERASE character
5100026 #define VINTR    3      // INTR character
5100027 #define VKILL    4      // KILL character
5100028 #define VMIN     5      // MIN value
5100029 #define VQUIT    6      // QUIT character
5100030 #define VSTART   7      // START character
5100031 #define VSTOP    8      // STOP character
5100032 #define VSUSP    9      // SUSP character
5100033 #define VTIME    10     // TIME value
5100034 //
5100035 // Input modes, for 'c_iflag' inside the 'termios'
5100036 // structure:
5100037 //
5100038 #define BRKINT    1      // signal interrupt on break
5100039 #define ICRNL    2      // map CR to NL on input
5100040 #define IGNBRK   4      // ignore break condition
5100041 #define IGNCR    8      // ignore CR
5100042 #define IGNPAR   16     // ignore characters with
5100043 // parity errors
5100044 #define INLCR    32     // map NL to CR on input
5100045 #define INPCK    64     // enable input parity check
5100046 #define ISTRIP   128    // strip off eighth bit
5100047 #define IXOFF    256    // enable start/stop input
5100048 // control
5100049 #define IXON     512    // enable start/stop output
5100050 // control
5100051 #define PARMRK   1024   // mark parity errors
5100052 //
5100053 // Output modes, for 'c_oflag' inside the 'termios'
5100054 // structure:
5100055 //
5100056 #define OPOST    1      // post-process output
5100057 //
5100058 // Control modes, for 'c_cflag' inside the 'termios'
```

```
510059 // structure:
510060 //   not implemented.
510061 //
510062 //
510063 // Local modes, for 'c_lflag' inside the 'termios'
510064 // structure:
510065 //
510066 #define ECHO      1      // enable echo
510067 #define ECHOE    2      // echo erase character as
510068                      // backspace
510069 #define ECHOK    4      // echo KILL
510070 #define ECHONL   8      // echo NL
510071 #define ICANON   16     // canonical input mode
510072 #define IEXTEN   32     // extended input mode
510073 #define ISIG     64     // enable signals
510074 #define NOFLSH   128    // disable flush after
510075                      // interrupt or quit
510076 #define TOSTOP   256    // send SIGTTOU for background
510077                      // output
510078 //-----
510079 // Optional action for use with 'tcsetattr()':
510080 //
510081 #define TCSANOW  1      // change attributes
510082                      // immediately
510083 #define TCSADRAIN 2     // change attributes when
510084                      // output has drained
510085 #define TCSAFLUSH 3    // change attributes when
510086                      // output has drained,
510087                      // and also flush pending
510088                      // input
510089 //-----
510090 int tcgetattr (int fdn, struct termios *termios_p);
510091 int tcsetattr (int fdn, int action,
510092              struct termios *termios_p);
510093 //-----
510094 #endif
```


[95.28.1 lib/termios/tcgetattr.c](#) [2161](#)

[95.28.2 lib/termios/tcsetattr.c](#) [2161](#)

95.28.1 lib/termios/tcgetattr.c



Si veda la sezione [87.58](#).

```
5110001 #include <termios.h>
5110002 #include <sys/os32.h>
5110003 #include <errno.h>
5110004 //-----
5110005 #define DEBUG 0
5110006 //-----
5110007 int
5110008 tcgetattr (int fdn, struct termios *termios_p)
5110009 {
5110010     sysmsg_tcatr_t msg;
5110011     msg.fdn = fdn;
5110012     msg.attr = termios_p;
5110013     sys (SYS_TCGETATTR, &msg, (sizeof msg));
5110014     errno = msg.errno;
5110015     errln = msg.errln;
5110016     strncpy (errfn, msg.errfn, PATH_MAX);
5110017     return (msg.ret);
5110018 }
```

95.28.2 lib/termios/tcsetattr.c



Si veda la sezione [87.58](#).

```
5120001 #include <termios.h>
5120002 #include <sys/os32.h>
5120003 #include <errno.h>
5120004 //-----
5120005 #define DEBUG 0
5120006 //-----
5120007 int
```

```
5120008 tcsetattr (int fdn, int action, struct termios *termios_p)
5120009 {
5120010     sysmsg_tcattr_t msg;
5120011     msg.fdn = fdn;
5120012     msg.action = action;
5120013     msg.attr = termios_p;
5120014     sys (SYS_TCSETATTR, &msg, (sizeof msg));
5120015     errno = msg.errno;
5120016     errln = msg.errln;
5120017     strncpy (errfn, msg.errfn, PATH_MAX);
5120018     return (msg.ret);
5120019 }
```

95.29 os32: «lib/time.h»

«

Si veda la sezione [91.3](#).

```
5130001 #ifndef _TIME_H
5130002 #define _TIME_H          1
5130003 //-----
5130004 #include <restrict.h>
5130005 #include <size_t.h>
5130006 #include <time_t.h>
5130007 #include <clock_t.h>
5130008 #include <NULL.h>
5130009 #include <stdint.h>
5130010 //-----
5130011 #define CLOCKS_PER_SEC ((clock_t) 100)
5130012 //-----
5130013 struct tm
5130014 {
5130015     int tm_sec;
5130016     int tm_min;
5130017     int tm_hour;
5130018     int tm_mday;
5130019     int tm_mon;
5130020     int tm_year;
```

```

5130021     int tm_wday;
5130022     int tm_yday;
5130023     int tm_isdst;
5130024 };
5130025 //-----
5130026 clock_t clock (void);
5130027 time_t time (time_t * timer);
5130028 int stime (time_t * timer);
5130029 double difftime (time_t time1, time_t time0);
5130030 time_t mktime (const struct tm *timeptr);
5130031 struct tm *gmtime (const time_t * timer);
5130032 struct tm *localtime (const time_t * timer);
5130033 char *asctime (const struct tm *timeptr);
5130034 char *ctime (const time_t * timer);
5130035 size_t strftime (char *restrict s, size_t maxsize,
5130036                 const char *restrict format,
5130037                 const struct tm *restrict timeptr);
5130038 //-----
5130039 #define difftime(t1,t0) ((double)((t1)-(t0)))
5130040 #define ctime(t)         (asctime (localtime (t)))
5130041 #define localtime(t)    (gmtime (t))
5130042 //-----
5130043 #endif

```

95.29.1	lib/time/asctime.c	2164
95.29.2	lib/time/clock.c	2166
95.29.3	lib/time/gmtime.c	2167
95.29.4	lib/time/mktime.c	2172
95.29.5	lib/time/stime.c	2176
95.29.6	lib/time/time.c	2177

95.29.1 lib/time/asctime.c



Si veda la sezione [88.15](#).

```
5140001 #include <time.h>
5140002 #include <string.h>
5140003 #include <stdio.h>
5140004 //-----
5140005 char *
5140006 asctime (const struct tm *timeptr)
5140007 {
5140008     static char time_string[25]; // 'Sun Jan 30
5140009     // 24:00:00 2111'
5140010     //
5140011     // Check argument.
5140012     //
5140013     if (timeptr == NULL)
5140014     {
5140015         return (NULL);
5140016     }
5140017     //
5140018     // Set week day.
5140019     //
5140020     switch (timeptr->tm_wday)
5140021     {
5140022     case 0:
5140023         strcpy (&time_string[0], "Sun");
5140024         break;
5140025     case 1:
5140026         strcpy (&time_string[0], "Mon");
5140027         break;
5140028     case 2:
5140029         strcpy (&time_string[0], "Tue");
5140030         break;
5140031     case 3:
5140032         strcpy (&time_string[0], "Wed");
5140033         break;
5140034     case 4:
```

```
5140035     strcpy (&time_string[0], "Thu");
5140036     break;
5140037     case 5:
5140038         strcpy (&time_string[0], "Fri");
5140039         break;
5140040     case 6:
5140041         strcpy (&time_string[0], "Sat");
5140042         break;
5140043     default:
5140044         strcpy (&time_string[0], "Err");
5140045     }
5140046     //
5140047     // Set month.
5140048     //
5140049     switch (timeptr->tm_mon)
5140050     {
5140051     case 1:
5140052         strcpy (&time_string[3], " Jan");
5140053         break;
5140054     case 2:
5140055         strcpy (&time_string[3], " Feb");
5140056         break;
5140057     case 3:
5140058         strcpy (&time_string[3], " Mar");
5140059         break;
5140060     case 4:
5140061         strcpy (&time_string[3], " Apr");
5140062         break;
5140063     case 5:
5140064         strcpy (&time_string[3], " May");
5140065         break;
5140066     case 6:
5140067         strcpy (&time_string[3], " Jun");
5140068         break;
5140069     case 7:
5140070         strcpy (&time_string[3], " Jul");
5140071         break;
```

```
5140072     case 8:
5140073         strcpy (&time_string[3], " Aug");
5140074         break;
5140075     case 9:
5140076         strcpy (&time_string[3], " Sep");
5140077         break;
5140078     case 10:
5140079         strcpy (&time_string[3], " Oct");
5140080         break;
5140081     case 11:
5140082         strcpy (&time_string[3], " Nov");
5140083         break;
5140084     case 12:
5140085         strcpy (&time_string[3], " Dec");
5140086         break;
5140087     default:
5140088         strcpy (&time_string[3], " Err");
5140089     }
5140090     //
5140091     // Set day of month, hour, minute, second and year.
5140092     //
5140093     sprintf (&time_string[7], " %2i %2i:%2i:%2i %4i",
5140094             timeptr->tm_mday, timeptr->tm_hour,
5140095             timeptr->tm_min, timeptr->tm_sec,
5140096             timeptr->tm_year);
5140097     //
5140098     //
5140099     //
5140100     return (&time_string[0]);
5140101 }
```

95.29.2 lib/time/clock.c



Si veda la sezione [87.9](#).

```
5150001 #include <time.h>
5150002 #include <sys/os32.h>
```

```
5150003 //-----
5150004 clock_t
5150005 clock (void)
5150006 {
5150007     sysmsg_clock_t msg;
5150008     msg.ret = 0;
5150009     sys (SYS_CLOCK, &msg, (sizeof msg));
5150010     return (msg.ret);
5150011 }
```

95.29.3 lib/time/gmtime.c

Si veda la sezione [88.15](#).



```
5160001 #include <time.h>
5160002 //-----
5160003 static int leap_year (int year);
5160004 //-----
5160005 struct tm *
5160006 gmtime (const time_t * timer)
5160007 {
5160008     static struct tm tms;
5160009     int loop;
5160010     unsigned int remainder;
5160011     unsigned int days;
5160012     //
5160013     // Check argument.
5160014     //
5160015     if (timer == NULL)
5160016     {
5160017         return (NULL);
5160018     }
5160019     //
5160020     // Days since epoch. There are 86400 seconds per
5160021     // day.
5160022     // At the moment, the field 'tm_yday' will contain
5160023     // all days since epoch.
```

```
5160024 //
5160025 days = *timer / 86400L;
5160026 remainder = *timer % 86400L;
5160027 //
5160028 // Minutes, after full days.
5160029 //
5160030 tms.tm_min = remainder / 60U;
5160031 //
5160032 // Seconds, after full minutes.
5160033 //
5160034 tms.tm_sec = remainder % 60U;
5160035 //
5160036 // Hours, after full days.
5160037 //
5160038 tms.tm_hour = tms.tm_min / 60;
5160039 //
5160040 // Minutes, after full hours.
5160041 //
5160042 tms.tm_min = tms.tm_min % 60;
5160043 //
5160044 // Find the week day. Must remove some days to align
5160045 // the
5160046 // calculation. So: the week days of the first week
5160047 // of 1970
5160048 // are not valid! After 1970-01-04 calculations are
5160049 // right.
5160050 //
5160051 tms.tm_wday = (days - 3) % 7;
5160052 //
5160053 // Find the year: the field 'tm_yday' will be
5160054 // reduced to the days
5160055 // of current year.
5160056 //
5160057 for (tms.tm_year = 1970; days > 0; tms.tm_year++)
5160058     {
5160059         if (leap_year (tms.tm_year))
5160060             {
```



```
5160061         if (days >= 366)
5160062             {
5160063                 days -= 366;
5160064                 continue;
5160065             }
5160066         else
5160067             {
5160068                 break;
5160069             }
5160070     }
5160071     else
5160072     {
5160073         if (days >= 365)
5160074             {
5160075                 days -= 365;
5160076                 continue;
5160077             }
5160078         else
5160079             {
5160080                 break;
5160081             }
5160082     }
5160083 }
5160084 //
5160085 // Day of the year.
5160086 //
5160087 tms.tm_yday = days + 1;
5160088 //
5160089 // Find the month.
5160090 //
5160091 tms.tm_mday = days + 1;
5160092 //
5160093 for (tms.tm_mon = 0, loop = 1; tms.tm_mon <= 12 && loop;)
5160094     {
5160095         tms.tm_mon++;
5160096         //
5160097         switch (tms.tm_mon)
```

```
5160098     {
5160099     case 1:
5160100     case 3:
5160101     case 5:
5160102     case 7:
5160103     case 8:
5160104     case 10:
5160105     case 12:
5160106         if (tms.tm_mday >= 31)
5160107             {
5160108                 tms.tm_mday -= 31;
5160109             }
5160110         else
5160111             {
5160112                 loop = 0;
5160113             }
5160114         break;
5160115     case 4:
5160116     case 6:
5160117     case 9:
5160118     case 11:
5160119         if (tms.tm_mday >= 30)
5160120             {
5160121                 tms.tm_mday -= 30;
5160122             }
5160123         else
5160124             {
5160125                 loop = 0;
5160126             }
5160127         break;
5160128     case 2:
5160129         if (leap_year (tms.tm_year))
5160130             {
5160131                 if (tms.tm_mday >= 29)
5160132                     {
5160133                         tms.tm_mday -= 29;
5160134                     }
```

```
5160135         else
5160136             {
5160137                 loop = 0;
5160138             }
5160139         }
5160140     else
5160141     {
5160142         if (tms.tm_mday >= 28)
5160143         {
5160144             tms.tm_mday -= 28;
5160145         }
5160146     else
5160147     {
5160148         loop = 0;
5160149     }
5160150 }
5160151 break;
5160152 }
5160153 }
5160154 //
5160155 // No check for day light saving time.
5160156 //
5160157 tms.tm_isdst = 0;
5160158 //
5160159 // Return.
5160160 //
5160161 return (&tms);
5160162 }
5160163
5160164 //-----
5160165 static int
5160166 leap_year (int year)
5160167 {
5160168     if ((year % 4) == 0)
5160169     {
5160170         if ((year % 100) == 0)
5160171         {
```

```
5160172         if ((year % 400) == 0)
5160173             {
5160174                 return (1);
5160175             }
5160176         else
5160177             {
5160178                 return (0);
5160179             }
5160180     }
5160181     else
5160182     {
5160183         return (1);
5160184     }
5160185 }
5160186 else
5160187 {
5160188     return (0);
5160189 }
5160190 }
```

95.29.4 lib/time/mktime.c

<<

Si veda la sezione [88.15](#).

```
5170001 #include <time.h>
5170002 #include <string.h>
5170003 #include <stdio.h>
5170004 //-----
5170005 static int leap_year (int year);
5170006 //-----
5170007 time_t
5170008 mktime (const struct tm *timeptr)
5170009 {
5170010     time_t timer_total;
5170011     time_t timer_aux;
5170012     int days;
5170013     int month;
```

```
5170014     int year;
5170015     //
5170016     // From seconds to days.
5170017     //
5170018     timer_total = timeptr->tm_sec;
5170019     //
5170020     timer_aux = timeptr->tm_min;
5170021     timer_aux *= 60;
5170022     timer_total += timer_aux;
5170023     //
5170024     timer_aux = timeptr->tm_hour;
5170025     timer_aux *= (60 * 60);
5170026     timer_total += timer_aux;
5170027     //
5170028     timer_aux = timeptr->tm_mday;
5170029     timer_aux *= 24;
5170030     timer_aux *= (60 * 60);
5170031     timer_total += timer_aux;
5170032     //
5170033     // Month: add the days of months.
5170034     // Will scan the months, from the first, but before
5170035     // the
5170036     // months of the value inside field 'tm_mon'.
5170037     //
5170038     for (month = 1, days = 0; month < timeptr->tm_mon;
5170039         month++)
5170040     {
5170041         switch (month)
5170042         {
5170043             case 1:
5170044             case 3:
5170045             case 5:
5170046             case 7:
5170047             case 8:
5170048             case 10:
5170049                 //
5170050                 // There is no December, because the scan
```

```
5170051         // can go up to
5170052         // the month before the value inside field
5170053         // 'tm_mon'.
5170054         //
5170055         days += 31;
5170056         break;
5170057     case 4:
5170058     case 6:
5170059     case 9:
5170060     case 11:
5170061         days += 30;
5170062         break;
5170063     case 2:
5170064         if (leap_year (timeptr->tm_year))
5170065             {
5170066                 days += 29;
5170067             }
5170068         else
5170069             {
5170070                 days += 28;
5170071             }
5170072         break;
5170073     }
5170074 }
5170075 //
5170076 timer_aux = days;
5170077 timer_aux *= 24;
5170078 timer_aux *= (60 * 60);
5170079 timer_total += timer_aux;
5170080 //
5170081 // Year. The work is similar to the one of months:
5170082 // days of
5170083 // years are counted, up to the year before the one
5170084 // reported
5170085 // by the field 'tm_year'.
5170086 //
5170087 for (year = 1970, days = 0; year < timeptr->tm_year;
```

```
5170088     year++)
5170089     {
5170090         if (leap_year (year))
5170091             {
5170092                 days += 366;
5170093             }
5170094         else
5170095             {
5170096                 days += 365;
5170097             }
5170098     }
5170099     //
5170100     // After all, must subtract a day from the total.
5170101     //
5170102     days--;
5170103     //
5170104     timer_aux = days;
5170105     timer_aux *= 24;
5170106     timer_aux *= (60 * 60);
5170107     timer_total += timer_aux;
5170108     //
5170109     // That's all.
5170110     //
5170111     return (timer_total);
5170112 }
5170113
5170114 //-----
5170115 int
5170116 leap_year (int year)
5170117 {
5170118     if ((year % 4) == 0)
5170119         {
5170120             if ((year % 100) == 0)
5170121                 {
5170122                     if ((year % 400) == 0)
5170123                         {
5170124                             return (1);
```

```
5170125         }
5170126         else
5170127         {
5170128             return (0);
5170129         }
5170130     }
5170131     else
5170132     {
5170133         return (1);
5170134     }
5170135 }
5170136 else
5170137 {
5170138     return (0);
5170139 }
5170140 }
```

95.29.5 lib/time/stime.c



Si veda la sezione [87.59](#).

```
5180001 #include <time.h>
5180002 #include <sys/os32.h>
5180003 #include <errno.h>
5180004 //-----
5180005 int
5180006 stime (time_t * timer)
5180007 {
5180008     sysmsg_stime_t msg;
5180009     //
5180010     if (timer == NULL)
5180011     {
5180012         errset (EINVAL);
5180013         return (-1);
5180014     }
5180015     //
5180016     msg.timer = *timer;
```



```

5180017     msg.ret = 0;
5180018     sys (SYS_STIME, &msg, (sizeof msg));
5180019     return (msg.ret);
5180020 }

```

95.29.6 lib/time/time.c

Si veda la sezione [87.59](#).

```

5190001 #include <time.h>
5190002 #include <sys/os32.h>
5190003 //-----
5190004 time_t
5190005 time (time_t * timer)
5190006 {
5190007     sysmsg_time_t msg;
5190008     msg.ret = ((time_t) 0);
5190009     sys (SYS_TIME, &msg, (sizeof msg));
5190010     if (timer != NULL)
5190011     {
5190012         *timer = msg.ret;
5190013     }
5190014     return (msg.ret);
5190015 }

```

95.30 os32: «lib/unistd.h»

Si veda la sezione [91.3](#).

```

5200001 #ifndef _UNISTD_H
5200002 #define _UNISTD_H         1
5200003 //-----
5200004 #include <sys/stat.h>
5200005 #include <sys/types.h> // size_t, ssize_t, uid_t,
5200006                       // gid_t, off_t, pid_t
5200007 #include <inttypes.h> // intptr_t

```

```
520008 #include <SEEK.h> // SEEK_CUR, SEEK_SET,
520009 // SEEK_END
520010 //-----
520011 typedef unsigned int useconds_t; // This type
520012 // should be
520013 // used for
520014 // the
520015 // obsolete function
520016 // 'usleep()', that
520017 // is only
520018 // implemented inside
520019 // the
520020 // kernel, as
520021 // 'k_usleep()', for
520022 // the
520023 // drivers
520024 // management.
520025 //-----
520026 extern char **environ; // Variable 'environ' is used
520027 // by functions like
520028 // 'execv()' in replacement
520029 // for 'envp[][]'.
520030 //-----
520031 extern char *optarg; // Used by 'optarg()'.
520032 extern int optind; //
520033 extern int opterr; //
520034 extern int optopt; //
520035 //-----
520036 #define STDIN_FILENO 0 //
520037 #define STDOUT_FILENO 1 // Standard file
520038 // descriptors.
520039 #define STDERR_FILENO 2 //
520040 //-----
520041 #define R_OK 4 // Read permission.
520042 #define W_OK 2 // Write permission.
520043 #define X_OK 1 // Execute or traverse
520044 // permission.
```

```
5200045 #define F_OK          0          // File exists.
5200046 //-----
5200047
5200048 int access (const char *path, int mode);
5200049 int brk (void *address);
5200050 int chdir (const char *path);
5200051 int chown (const char *path, uid_t uid, gid_t gid);
5200052 int close (int fdn);
5200053 int dup (int fdn_old);
5200054 int dup2 (int fdn_old, int fdn_new);
5200055 int execl (const char *path, char *arg, ...);
5200056 int execle (const char *path, char *arg, ...);
5200057 int execlp (const char *path, char *arg, ...);
5200058 int execv (const char *path, char *const argv[]);
5200059 int execve (const char *path, char *const argv[],
5200060             char *const envp[]);
5200061 int execvp (const char *path, char *const argv[]);
5200062 void _exit (int status);
5200063 int fchown (int fdn, uid_t uid, gid_t gid);
5200064 pid_t fork (void);
5200065 char *getcwd (char *buffer, size_t size);
5200066 gid_t getegid (void);
5200067 uid_t geteuid (void);
5200068 gid_t getgid (void);
5200069 int getopt (int argc, char *const argv[],
5200070             const char *optstring);
5200071 pid_t getpgrp (void);
5200072 pid_t getppid (void);
5200073 pid_t getpid (void);
5200074 uid_t getuid (void);
5200075 int isatty (int fdn);
5200076 int link (const char *path_old, const char *path_new);
5200077 off_t lseek (int fdn, off_t offset, int whence);
5200078 #define      nice(n)      (0)
5200079 int pipe (int pipefd[2]);
5200080 ssize_t read (int fdn, void *buffer, size_t count);
5200081 #define      readlink(p,b,s) ((ssize_t) -1)
```

```

5200082 int rmdir (const char *path);
5200083 void *sbrk (intptr_t increment);
5200084 int setegid (gid_t gid);
5200085 int seteuid (uid_t uid);
5200086 int setgid (gid_t gid);
5200087 int setpgrp (void);
5200088 int setuid (uid_t uid);
5200089 unsigned int sleep (unsigned int s);
5200090 #define      sync() /**/
5200091 char *ttyname (int fdn);
5200092 int unlink (const char *path);
5200093 ssize_t write (int fdn, const void *buffer, size_t count);
5200094 //-----
5200095 #endif

```

95.30.1	lib/unistd/_exit.c	2182
95.30.2	lib/unistd/access.c	2183
95.30.3	lib/unistd/brk.c	2184
95.30.4	lib/unistd/chdir.c	2185
95.30.5	lib/unistd/chown.c	2186
95.30.6	lib/unistd/close.c	2187
95.30.7	lib/unistd/dup.c	2187
95.30.8	lib/unistd/dup2.c	2188
95.30.9	lib/unistd/environ.c	2189
95.30.10	lib/unistd/execl.c	2189
95.30.11	lib/unistd/execl.c	2190
95.30.12	lib/unistd/execlp.c	2191
95.30.13	lib/unistd/execv.c	2193

Sorgenti della libreria generale	2181
95.30.14 lib/unistd/execve.c	2193
95.30.15 lib/unistd/execvp.c	2196
95.30.16 lib/unistd/fchdir.c	2197
95.30.17 lib/unistd/fchown.c	2197
95.30.18 lib/unistd/fork.c	2198
95.30.19 lib/unistd/getcwd.c	2199
95.30.20 lib/unistd/getegid.c	2201
95.30.21 lib/unistd/geteuid.c	2201
95.30.22 lib/unistd/getgid.c	2202
95.30.23 lib/unistd/getopt.c	2202
95.30.24 lib/unistd/getpgrp.c	2209
95.30.25 lib/unistd/getpid.c	2210
95.30.26 lib/unistd/getppid.c	2210
95.30.27 lib/unistd/getuid.c	2211
95.30.28 lib/unistd/isatty.c	2211
95.30.29 lib/unistd/link.c	2213
95.30.30 lib/unistd/lseek.c	2213
95.30.31 lib/unistd/pipe.c	2214
95.30.32 lib/unistd/read.c	2215
95.30.33 lib/unistd/rmdir.c	2218
95.30.34 lib/unistd/sbrk.c	2219
95.30.35 lib/unistd/setegid.c	2220

95.30.36	lib/unistd/seteuid.c	2220
95.30.37	lib/unistd/setgid.c	2221
95.30.38	lib/unistd/setpgrp.c	2222
95.30.39	lib/unistd/setuid.c	2222
95.30.40	lib/unistd/sleep.c	2223
95.30.41	lib/unistd/ttyname.c	2224
95.30.42	lib/unistd/unlink.c	2226
95.30.43	lib/unistd/write.c	2226

95.30.1 lib/unistd/_exit.c

«

Si veda la sezione [87.2](#).

```
5210001 #include <unistd.h>
5210002 #include <sys/os32.h>
5210003 //-----
5210004 void
5210005 _exit (int status)
5210006 {
5210007     sysmsg_exit_t msg;
5210008     //
5210009     // Only the low eight bit are returned.
5210010     //
5210011     msg.status = (status & 0xFF);
5210012     //
5210013     //
5210014     //
5210015     sys (SYS_EXIT, &msg, (sizeof msg));
5210016     //
5210017     // Should not return from system call, but if it
5210018     // does, loop
5210019     // forever:
```

```
5210020 //
5210021 while (1);
5210022 }
```

95.30.2 lib/unistd/access.c



Si veda la sezione [88.4](#).

```
5220001 #include <unistd.h>
5220002 #include <sys/stat.h>
5220003 #include <errno.h>
5220004 //-----
5220005 int
5220006 access (const char *path, int mode)
5220007 {
5220008     struct stat st;
5220009     int status;
5220010     uid_t euid;
5220011     //
5220012     status = stat (path, &st);
5220013     if (status != 0)
5220014     {
5220015         return (-1);
5220016     }
5220017     //
5220018     // File exists?
5220019     //
5220020     if (mode == F_OK)
5220021     {
5220022         return (0);
5220023     }
5220024     //
5220025     // Some access permissions are requested: get
5220026     // effective user id.
5220027     //
5220028     euid = geteuid ();
5220029     //
```

```
5220030 // Check owner access permissions.
5220031 //
5220032 if (st.st_uid == euid
5220033     && ((st.st_mode & S_IRWXU) == (mode << 6)))
5220034     {
5220035         return (0);
5220036     }
5220037 //
5220038 // Check others access permissions.
5220039 //
5220040 if ((st.st_mode & S_IRWXO) == (mode))
5220041     {
5220042         return (0);
5220043     }
5220044 //
5220045 // Otherwise there are no access permissions.
5220046 //
5220047 errset (EACCES); // Permission denied.
5220048 return (-1);
5220049 }
```

95.30.3 lib/unistd/brk.c



Si veda la sezione [87.5](#).

```
5230001 #include <unistd.h>
5230002 #include <string.h>
5230003 #include <sys/os32.h>
5230004 #include <errno.h>
5230005 #include <limits.h>
5230006 //-----
5230007 int
5230008 brk (void *address)
5230009 {
5230010     sysmsg_brk_t msg;
5230011     //
5230012     if (address == NULL)
```



```
5230013     {
5230014         errset (EINVAL);
5230015         return (-1);
5230016     }
5230017     //
5230018     msg.address = address;
5230019     //
5230020     sys (SYS_BRK, &msg, (sizeof msg));
5230021     //
5230022     errno = msg.errno;
5230023     errln = msg.errln;
5230024     strncpy (errfn, msg.errfn, PATH_MAX);
5230025     return (msg.ret);
5230026 }
```

95.30.4 lib/unistd/chdir.c

Si veda la sezione [87.6](#).

```
5240001 #include <unistd.h>
5240002 #include <string.h>
5240003 #include <sys/os32.h>
5240004 #include <errno.h>
5240005 #include <limits.h>
5240006 //-----
5240007 int
5240008 chdir (const char *path)
5240009 {
5240010     sysmsg_chdir_t msg;
5240011     //
5240012     msg.path = path;
5240013     msg.ret = 0;
5240014     msg.errno = 0;
5240015     //
5240016     sys (SYS_CHDIR, &msg, (sizeof msg));
5240017     //
5240018     errno = msg.errno;
```

```
5240019     errln = msg.errln;
5240020     strncpy (errfn, msg.errfn, PATH_MAX);
5240021     return (msg.ret);
5240022 }
```

95.30.5 lib/unistd/chown.c

<<

Si veda la sezione [87.8](#).

```
5250001 #include <unistd.h>
5250002 #include <string.h>
5250003 #include <sys/os32.h>
5250004 #include <errno.h>
5250005 #include <limits.h>
5250006 //-----
5250007 int
5250008 chown (const char *path, uid_t uid, gid_t gid)
5250009 {
5250010     sysmsg_chown_t msg;
5250011     //
5250012     msg.path = path;
5250013     msg.uid = uid;
5250014     msg.gid = gid;
5250015     //
5250016     sys (SYS_CHOWN, &msg, (sizeof msg));
5250017     //
5250018     errno = msg.errno;
5250019     errln = msg.errln;
5250020     strncpy (errfn, msg.errfn, PATH_MAX);
5250021     return (msg.ret);
5250022 }
```

95.30.6 lib/unistd/close.c



Si veda la sezione [87.10](#).

```
5260001 #include <unistd.h>
5260002 #include <errno.h>
5260003 #include <sys/os32.h>
5260004 #include <string.h>
5260005 //-----
5260006 int
5260007 close (int fdn)
5260008 {
5260009     sysmsg_close_t msg;
5260010     msg.fdn = fdn;
5260011     //
5260012     while (1)
5260013     {
5260014         sys (SYS_CLOSE, &msg, (sizeof msg));
5260015         if (msg.ret < 0 && (msg.errno == EINPROGRESS
5260016             || msg.errno == EALREADY))
5260017             {
5260018                 continue;
5260019             }
5260020         //
5260021         break;
5260022     }
5260023     errno = msg.errno;
5260024     errln = msg.errln;
5260025     strncpy (errfn, msg.errfn, PATH_MAX);
5260026     return (msg.ret);
5260027 }
```

95.30.7 lib/unistd/dup.c



Si veda la sezione [87.12](#).

```
5270001 #include <unistd.h>
5270002 #include <sys/os32.h>
```

```
5270003 #include <string.h>
5270004 #include <errno.h>
5270005 //-----
5270006 int
5270007 dup (int fdn_old)
5270008 {
5270009     sysmsg_dup_t msg;
5270010     //
5270011     msg.fdn_old = fdn_old;
5270012     //
5270013     sys (SYS_DUP, &msg, (sizeof msg));
5270014     //
5270015     errno = msg.errno;
5270016     errln = msg.errln;
5270017     strncpy (errfn, msg.errfn, PATH_MAX);
5270018     return (msg.ret);
5270019 }
```

95.30.8 lib/unistd/dup2.c

<<

Si veda la sezione [87.12](#).

```
5280001 #include <unistd.h>
5280002 #include <sys/os32.h>
5280003 #include <string.h>
5280004 #include <errno.h>
5280005 //-----
5280006 int
5280007 dup2 (int fdn_old, int fdn_new)
5280008 {
5280009     sysmsg_dup2_t msg;
5280010     //
5280011     msg.fdn_old = fdn_old;
5280012     msg.fdn_new = fdn_new;
5280013     //
5280014     sys (SYS_DUP2, &msg, (sizeof msg));
5280015     //
```

```

5280016     errno = msg.errno;
5280017     errln = msg.errln;
5280018     strncpy (errfn, msg.errfn, PATH_MAX);
5280019     return (msg.ret);
5280020 }

```

95.30.9 lib/unistd/environ.c

Si veda la sezione [91.1](#).

```

5290001 #include <unistd.h>
5290002 //-----
5290003 char **environ;

```

95.30.10 lib/unistd/execl.c

Si veda la sezione [88.21](#).

```

5300001 #include <unistd.h>
5300002 #include <limits.h>
5300003 #include <stdarg.h>
5300004 #include <stddef.h>
5300005 //-----
5300006 int
5300007 execl (const char *path, char *arg, ...)
5300008 {
5300009     int argc;
5300010     char *arg_next;
5300011     char *argv[ARG_MAX / 2];
5300012     //
5300013     va_list ap;
5300014     va_start (ap, arg);
5300015     //
5300016     arg_next = arg;
5300017     //
5300018     for (argc = 0; argc < ARG_MAX / 2; argc++)
5300019     {

```

```

5300020     argv[argc] = arg_next;
5300021     if (argv[argc] == NULL)
5300022     {
5300023         break;           // End of arguments.
5300024     }
5300025     arg_next = va_arg (ap, char *);
5300026 }
5300027 //
5300028     return (execve (path, argv, environ));           // [1]
5300029 }
5300030
5300031 //
5300032 // The variable 'environ' is declared as
5300033 // 'char **environ' and is
5300034 // included from <unistd.h>.
5300035 //

```

95.30.11 lib/unistd/execl.c



Si veda la sezione [88.21](#).

```

5310001 #include <unistd.h>
5310002 #include <limits.h>
5310003 #include <stdarg.h>
5310004 #include <stddef.h>
5310005 //-----
5310006 int
5310007 execl (const char *path, char *arg, ...)
5310008 {
5310009     int argc;
5310010     char *arg_next;
5310011     char *argv[ARG_MAX / 2];
5310012     char **envp;
5310013     //
5310014     va_list ap;
5310015     va_start (ap, arg);
5310016     //

```

```

5310017     arg_next = arg;
5310018     //
5310019     for (argc = 0; argc < ARG_MAX / 2; argc++)
5310020     {
5310021         argv[argc] = arg_next;
5310022         if (argv[argc] == NULL)
5310023         {
5310024             break;           // End of arguments.
5310025         }
5310026         arg_next = va_arg (ap, char *);
5310027     }
5310028     //
5310029     envp = va_arg (ap, char **);
5310030     //
5310031     return (execve (path, argv, envp));
5310032 }

```

95.30.12 lib/unistd/execlp.c

Si veda la sezione [88.21](#).

```

5320001 #include <unistd.h>
5320002 #include <string.h>
5320003 #include <stdlib.h>
5320004 #include <errno.h>
5320005 #include <sys/os32.h>
5320006 //-----
5320007 int
5320008 execlp (const char *path, char *arg, ...)
5320009 {
5320010     int argc;
5320011     char *arg_next;
5320012     char *argv[ARG_MAX / 2];
5320013     char command[PATH_MAX];
5320014     int status;
5320015     //
5320016     va_list ap;

```

```
5320017 va_start (ap, arg);
5320018 //
5320019 arg_next = arg;
5320020 //
5320021 for (argc = 0; argc < ARG_MAX / 2; argc++)
5320022 {
5320023     argv[argc] = arg_next;
5320024     if (argv[argc] == NULL)
5320025     {
5320026         break;           // End of arguments.
5320027     }
5320028     arg_next = va_arg (ap, char *);
5320029 }
5320030 //
5320031 // Get a full command path if necessary.
5320032 //
5320033 status = namep (path, command, (size_t) PATH_MAX);
5320034 if (status != 0)
5320035 {
5320036     //
5320037     // Variable 'errno' is already set by
5320038     // 'commandp()'.
5320039     //
5320040     return (-1);
5320041 }
5320042 //
5320043 // Return calling 'execve()'
5320044 //
5320045 return (execve (command, argv, environ)); // [1]
5320046 }
5320047
5320048 //
5320049 // The variable 'environ' is declared as
5320050 // 'char **environ' and is
5320051 // included from <unistd.h>.
5320052 //
```


95.30.13 lib/unistd/execv.c



Si veda la sezione [88.21](#).

```
5330001 #include <unistd.h>
5330002 //-----
5330003 int
5330004 execv (const char *path, char *const argv[])
5330005 {
5330006     return (execve (path, argv, environ));           // [1]
5330007 }
5330008
5330009 //
5330010 // The variable 'environ' is declared as
5330011 // 'char **environ' and is
5330012 // included from <unistd.h>.
5330013 //
```

95.30.14 lib/unistd/execve.c



Si veda la sezione [87.14](#).

```
5340001 #include <unistd.h>
5340002 #include <sys/types.h>
5340003 #include <sys/os32.h>
5340004 #include <errno.h>
5340005 #include <string.h>
5340006 #include <string.h>
5340007 //-----
5340008 int
5340009 execve (const char *path, char *const argv[],
5340010        char *const envp[])
5340011 {
5340012     sysmsg_exec_t msg;
5340013     size_t size;
5340014     size_t arg_size;
5340015     int argc;
5340016     size_t env_size;
```

```
5340017 int envc;
5340018 char *arg_data = msg.arg_data;
5340019 char *env_data = msg.env_data;
5340020 //
5340021 msg.path = path;
5340022 msg.ret = 0;
5340023 msg.errno = 0;
5340024 //
5340025 // Copy 'argv[]' inside a the message buffer
5340026 // 'msg.arg_data',
5340027 // separating each string with a null character and
5340028 // counting the
5340029 // number of strings inside 'argc'.
5340030 //
5340031 for (argc = 0, arg_size = 0, size = 0;
5340032      argv != NULL &&
5340033      argc < (ARG_MAX / 16) &&
5340034      arg_size < ARG_MAX / 2 &&
5340035      argv[argc] != NULL; argc++, arg_size += size)
5340036     {
5340037         size = strlen (argv[argc]);
5340038         size++; // Count also the final null
5340039         // character.
5340040         if (size > (ARG_MAX / 2 - arg_size))
5340041             {
5340042                 errset (E2BIG); // Argument list too
5340043                 // long.
5340044                 return (-1);
5340045             }
5340046         strncpy (arg_data, argv[argc], size);
5340047         arg_data += size;
5340048     }
5340049 msg.argc = argc;
5340050 //
5340051 // Copy 'envp[]' inside a the message buffer
5340052 // 'msg.env_data',
5340053 // separating each string with a null character and
```

```
5340054 // counting the
5340055 // number of strings inside 'envc'.
5340056 //
5340057 for (envc = 0, env_size = 0, size = 0;
5340058     envp != NULL &&
5340059     envc < (ARG_MAX / 16) &&
5340060     env_size < ARG_MAX / 2 &&
5340061     envp[envc] != NULL; envc++, env_size += size)
5340062     {
5340063     size = strlen (envp[envc]);
5340064     size++; // Count also the final null
5340065     // character.
5340066     if (size > (ARG_MAX / 2 - env_size))
5340067         {
5340068         errset (E2BIG); // Argument list too
5340069         // long.
5340070         return (-1);
5340071         }
5340072     strncpy (env_data, envp[envc], size);
5340073     env_data += size;
5340074     }
5340075 msg.envc = envc;
5340076 //
5340077 // System call.
5340078 //
5340079 sys (SYS_EXEC, &msg, (sizeof msg));
5340080 //
5340081 // Should not return, but if it does, then there is
5340082 // an error.
5340083 //
5340084 errno = msg.errno;
5340085 errln = msg.errln;
5340086 strncpy (errfn, msg.errfn, PATH_MAX);
5340087 return (msg.ret);
5340088 }
```

95.30.15 lib/unistd/execvp.c



Si veda la sezione [88.21](#).

```
5350001 #include <unistd.h>
5350002 #include <string.h>
5350003 #include <stdlib.h>
5350004 #include <errno.h>
5350005 #include <sys/os32.h>
5350006 //-----
5350007 int
5350008 execvp (const char *path, char *const argv[])
5350009 {
5350010     char command[PATH_MAX];
5350011     int status;
5350012     //
5350013     // Get a full command path if necessary.
5350014     //
5350015     status = namep (path, command, (size_t) PATH_MAX);
5350016     if (status != 0)
5350017     {
5350018         //
5350019         // Variable 'errno' is already set by 'namep()'.
5350020         //
5350021         return (-1);
5350022     }
5350023     //
5350024     // Return calling 'execve()'
5350025     //
5350026     return (execve (command, argv, environ));    // [1]
5350027 }
5350028
5350029 //
5350030 // The variable 'environ' is declared as
5350031 // 'char **environ' and is
5350032 // included from <unistd.h>.
5350033 //
```

95.30.16 lib/unistd/fchdir.c



Si veda la sezione [87.6](#).

```
5360001 #include <unistd.h>
5360002 #include <errno.h>
5360003 //-----
5360004 int
5360005 fchdir (int fdn)
5360006 {
5360007     //
5360008     // os32 requires to keep track of the path for the
5360009     // current working
5360010     // directory. The standard function 'fchdir()' is
5360011     // not applicable.
5360012     //
5360013     errset (E_NOT_IMPLEMENTED);
5360014     return (-1);
5360015 }
```

95.30.17 lib/unistd/fchown.c



Si veda la sezione [87.8](#).

```
5370001 #include <unistd.h>
5370002 #include <string.h>
5370003 #include <sys/os32.h>
5370004 #include <errno.h>
5370005 #include <limits.h>
5370006 //-----
5370007 int
5370008 fchown (int fdn, uid_t uid, gid_t gid)
5370009 {
5370010     sysmsg_fchown_t msg;
5370011     //
5370012     msg.fdn = fdn;
5370013     msg.uid = uid;
5370014     msg.gid = gid;
```

```
5370015 //
5370016 sys (SYS_FCHOWN, &msg, (sizeof msg));
5370017 //
5370018 errno = msg.errno;
5370019 errln = msg.errln;
5370020 strncpy (errfn, msg.errfn, PATH_MAX);
5370021 return (msg.ret);
5370022 }
```

95.30.18 lib/unistd/fork.c



Si veda la sezione [87.19](#).

```
5380001 #include <unistd.h>
5380002 #include <sys/types.h>
5380003 #include <sys/os32.h>
5380004 #include <errno.h>
5380005 #include <string.h>
5380006 //-----
5380007 pid_t
5380008 fork (void)
5380009 {
5380010     sysmsg_fork_t msg;
5380011     //
5380012     // Set the return value for the child process.
5380013     //
5380014     msg.ret = 0;
5380015     //
5380016     // Do the system call.
5380017     //
5380018     sys (SYS_FORK, &msg, (sizeof msg));
5380019     //
5380020     // If the system call has successfully generated a
5380021     // copy of
5380022     // the original process, the following code is
5380023     // executed from
5380024     // the parent and the child. But the child has the
```

```

5380025 // 'msg'
5380026 // structure untouched, while the parent has, at
5380027 // least, the
5380028 // pid number inside 'msg.ret'.
5380029 // If the system call fails, there is no child, and
5380030 // the
5380031 // parent finds the return value equal to -1, with
5380032 // an
5380033 // error number.
5380034 //
5380035 errno = msg.errno;
5380036 errln = msg.errln;
5380037 strncpy (errfn, msg.errfn, PATH_MAX);
5380038 return (msg.ret);
5380039 }

```

95.30.19 lib/unistd/getcwd.c

Si veda la sezione [87.21](#).

```

5390001 #include <unistd.h>
5390002 #include <sys/types.h>
5390003 #include <sys/os32.h>
5390004 #include <errno.h>
5390005 #include <stddef.h>
5390006 #include <string.h>
5390007 //-----
5390008 char *
5390009 getcwd (char *buffer, size_t size)
5390010 {
5390011     sysmsg_uarea_t msg;
5390012     //
5390013     // Check arguments: the buffer must be given.
5390014     //
5390015     if (buffer == NULL)
5390016     {
5390017         errset (EINVAL);

```

```
5390018     return (NULL);
5390019     }
5390020     //
5390021     msg.path_cwd = buffer;
5390022     msg.path_cwd_size = size;
5390023     //
5390024     // Set the last buffer element to zero, for later
5390025     // verification.
5390026     //
5390027     buffer[size - 1] = 0;
5390028     //
5390029     // Just get the user area data.
5390030     //
5390031     sys (SYS_UAREA, &msg, (sizeof msg));
5390032     //
5390033     // Check that the path is still correctly
5390034     // terminated. If it isn't,
5390035     // the path is longer than the buffer size, because
5390036     // the last null
5390037     // character was overwritten.
5390038     //
5390039     if (buffer[size - 1] != 0)
5390040     {
5390041         errset (ERANGE);
5390042         return (NULL);
5390043     }
5390044     //
5390045     // Everything is fine.
5390046     //
5390047     return (buffer);
5390048 }
```


95.30.20 lib/unistd/getegid.c



Si veda la sezione [87.22](#).

```
5400001 #include <unistd.h>
5400002 #include <sys/types.h>
5400003 #include <sys/os32.h>
5400004 #include <errno.h>
5400005 //-----
5400006 gid_t
5400007 getegid (void)
5400008 {
5400009     sysmsg_uarea_t msg;
5400010     msg.path_cwd = NULL;
5400011     msg.path_cwd_size = 0;
5400012     sys (SYS_UAREA, &msg, (sizeof msg));
5400013     return (msg.egid);
5400014 }
```

95.30.21 lib/unistd/geteuid.c



Si veda la sezione [87.27](#).

```
5410001 #include <unistd.h>
5410002 #include <sys/types.h>
5410003 #include <sys/os32.h>
5410004 #include <errno.h>
5410005 //-----
5410006 uid_t
5410007 geteuid (void)
5410008 {
5410009     sysmsg_uarea_t msg;
5410010     msg.path_cwd = NULL;
5410011     msg.path_cwd_size = 0;
5410012     sys (SYS_UAREA, &msg, (sizeof msg));
5410013     return (msg.euid);
5410014 }
```

95.30.22 lib/unistd/getgid.c



Si veda la sezione [87.22](#).

```
5420001 #include <unistd.h>
5420002 #include <sys/types.h>
5420003 #include <sys/os32.h>
5420004 #include <errno.h>
5420005 //-----
5420006 gid_t
5420007 getgid (void)
5420008 {
5420009     sysmsg_uarea_t msg;
5420010     msg.path_cwd = NULL;
5420011     msg.path_cwd_size = 0;
5420012     sys (SYS_UAREA, &msg, (sizeof msg));
5420013     return (msg.gid);
5420014 }
```

95.30.23 lib/unistd/getopt.c



Si veda la sezione [88.56](#).

```
5430001 #include <unistd.h>
5430002 #include <sys/types.h>
5430003 #include <sys/os32.h>
5430004 #include <errno.h>
5430005 //-----
5430006 char *optarg;
5430007 int optind = 1;
5430008 int opterr = 1;
5430009 int optopt = 0;
5430010 //-----
5430011 static void getopt_no_argument (int opt);
5430012 //-----
5430013 int
5430014 getopt (int argc, char *const argv[], const char *optstring)
5430015 {
```

```
5430016 static int o = 0; // Index to scan grouped
5430017 // options.
5430018 int s; // Index to scan 'optstring'
5430019 int opt; // Current option letter.
5430020 int flag_argument; // If there should be an
5430021 // argument.
5430022 //
5430023 // Entering the function, 'flag_argument' is zero.
5430024 // Just to make
5430025 // it clear:
5430026 //
5430027 flag_argument = 0;
5430028 //
5430029 // Scan 'argv[]' elements, starting from the value
5430030 // that 'optind'
5430031 // already have.
5430032 //
5430033 for (; optind < argc; optind++)
5430034 {
5430035 //
5430036 // If an option is expected, some check must be
5430037 // done at
5430038 // the beginning.
5430039 //
5430040 if (!flag_argument)
5430041 {
5430042 //
5430043 // Check if the scan is finished and
5430044 // 'optind' should be kept
5430045 // untouched:
5430046 // 'argv[optind]' is a null pointer;
5430047 // 'argv[optind][0]' is not the character
5430048 // '-';
5430049 // 'argv[optind]' points to the string "-";
5430050 // all 'argv[]' elements are parsed.
5430051 //
5430052 if (argv[optind] == NULL
```

```
5430053     || argv[optind][0] != '-'
5430054     || argv[optind][1] == 0 || optind >= argc)
5430055     {
5430056         return (-1);
5430057     }
5430058     //
5430059     // Check if the scan is finished and
5430060     // 'optind' is to be
5430061     // incremented:
5430062     // 'argv[optind]' points to the string "--".
5430063     //
5430064     if (argv[optind][0] == '-'
5430065         && argv[optind][1] == '-'
5430066         && argv[optind][2] == 0)
5430067     {
5430068         optind++;
5430069         return (-1);
5430070     }
5430071 }
5430072 //
5430073 // Scan 'argv[optind]' using the static index
5430074 // 'o'.
5430075 //
5430076 for (; o < strlen (argv[optind]); o++)
5430077 {
5430078     //
5430079     // If there should be an option, index 'o'
5430080     // should
5430081     // start from 1, because 'argv[optind][0]'
5430082     // must
5430083     // be equal to '-'.
5430084     //
5430085     if (!flag_argument && (o == 0))
5430086     {
5430087         //
5430088         // As there is no options, 'o' cannot
5430089         // start
```

```
5430090         // from zero, so a new loop is done.
5430091         //
5430092         continue;
5430093     }
5430094     //
5430095     if (flag_argument)
5430096     {
5430097         //
5430098         // There should be an argument, starting
5430099         // from
5430100         // `argv[optind][o]`.
5430101         //
5430102         if ((o == 0) && (argv[optind][o] == '-'))
5430103         {
5430104             //
5430105             // `argv[optind][0]` is equal to
5430106             // '-', but there
5430107             // should be an argument instead:
5430108             // the argument
5430109             // is missing.
5430110             //
5430111             optarg = NULL;
5430112             //
5430113             if (optstring[0] == ':')
5430114             {
5430115                 //
5430116                 // As the option string starts
5430117                 // with ':' the
5430118                 // function must return ':'.
5430119                 //
5430120                 optopt = opt;
5430121                 opt = ':';
5430122             }
5430123             else
5430124             {
5430125                 //
5430126                 // As the option string does not
```

```
5430127         // start with ':'
5430128         // the function must return '?'.
5430129         //
5430130         getopt_no_argument (opt);
5430131         optopt = opt;
5430132         opt = '?';
5430133     }
5430134     //
5430135     // 'optind' is left untouched.
5430136     //
5430137 }
5430138 else
5430139 {
5430140     //
5430141     // The argument is found: 'optind'
5430142     // is to be
5430143     // incremented and 'o' is reset.
5430144     //
5430145     optarg = &argv[optind][o];
5430146     optind++;
5430147     o = 0;
5430148 }
5430149 //
5430150 // Return the option, or ':', or '?'.
5430151 //
5430152 return (opt);
5430153 }
5430154 else
5430155 {
5430156     //
5430157     // It should be an option: 'optstring[]'
5430158     // must be
5430159     // scanned.
5430160     //
5430161     opt = argv[optind][o];
5430162     //
5430163     for (s = 0, optopt = 0;
```

```
5430164         s < strlen (optstring); s++)
5430165     {
5430166         //
5430167         // If 'optsting[0]' is equal to ':',
5430168         // index 's' must
5430169         // start at 1.
5430170         //
5430171         if ((s == 0) && (optstring[0] == ':'))
5430172             {
5430173                 continue;
5430174             }
5430175         //
5430176         if (opt == optstring[s])
5430177             {
5430178                 //
5430179                 if (optstring[s + 1] == ':')
5430180                     {
5430181                         //
5430182                         // There is an argument.
5430183                         //
5430184                         flag_argument = 1;
5430185                         break;
5430186                     }
5430187                 else
5430188                     {
5430189                         //
5430190                         // There is no argument.
5430191                         //
5430192                         o++;
5430193                         return (opt);
5430194                     }
5430195             }
5430196     }
5430197     //
5430198     if (s >= strlen (optstring))
5430199     {
5430200         //
```

```
5430201         // The 'optstring' scan is concluded
5430202         // with no
5430203         // match.
5430204         //
5430205         o++;
5430206         optopt = opt;
5430207         return ('?');
5430208     }
5430209     //
5430210     // Otherwise the loop was broken.
5430211     //
5430212 }
5430213 }
5430214 //
5430215 // Check index 'o'.
5430216 //
5430217 if (o >= strlen (argv[optind]))
5430218 {
5430219     //
5430220     // There are no more options or there is no
5430221     // argument
5430222     // inside current 'argv[optind]' string.
5430223     // Index 'o' is
5430224     // reset before the next loop.
5430225     //
5430226     o = 0;
5430227 }
5430228 }
5430229 //
5430230 // No more elements inside 'argv' or loop broken:
5430231 // there might be a
5430232 // missing argument.
5430233 //
5430234 if (flag_argument)
5430235 {
5430236     //
5430237     // Missing option argument.
```



```

5430238      //
5430239      optarg = NULL;
5430240      //
5430241      if (optstring[0] == ':')
5430242          {
5430243              return (':');
5430244          }
5430245      else
5430246          {
5430247              getopt_no_argument (opt);
5430248              return ('?');
5430249          }
5430250      }
5430251      //
5430252      return (-1);
5430253  }
5430254
5430255  //-----
5430256  static void
5430257  getopt_no_argument (int opt)
5430258  {
5430259      if (opterr)
5430260          {
5430261              fprintf (stderr,
5430262                      "Missing argument for option `-%c'\n", opt);
5430263          }
5430264  }

```

95.30.24 lib/unistd/getpgrp.c

Si veda la sezione [87.25](#).

```

5440001  #include <unistd.h>
5440002  #include <sys/types.h>
5440003  #include <sys/os32.h>
5440004  #include <errno.h>
5440005  //-----

```



```
5440006 pid_t
5440007 getpgrp (void)
5440008 {
5440009     sysmsg_uarea_t msg;
5440010     msg.path_cwd = NULL;
5440011     msg.path_cwd_size = 0;
5440012     sys (SYS_UAREA, &msg, (sizeof msg));
5440013     return (msg.pgrp);
5440014 }
```

95.30.25 lib/unistd/getpid.c



Si veda la sezione [87.25](#).

```
5450001 #include <unistd.h>
5450002 #include <sys/types.h>
5450003 #include <sys/os32.h>
5450004 #include <errno.h>
5450005 //-----
5450006 pid_t
5450007 getpid (void)
5450008 {
5450009     sysmsg_uarea_t msg;
5450010     msg.path_cwd = NULL;
5450011     msg.path_cwd_size = 0;
5450012     sys (SYS_UAREA, &msg, (sizeof msg));
5450013     return (msg.pid);
5450014 }
```

95.30.26 lib/unistd/getppid.c



Si veda la sezione [87.25](#).

```
5460001 #include <unistd.h>
5460002 #include <sys/types.h>
5460003 #include <sys/os32.h>
5460004 #include <errno.h>
```

```
5460005 //-----
5460006 pid_t
5460007 getppid (void)
5460008 {
5460009     sysmsg_uarea_t msg;
5460010     msg.path_cwd = NULL;
5460011     msg.path_cwd_size = 0;
5460012     sys (SYS_UAREA, &msg, (sizeof msg));
5460013     return (msg.ppid);
5460014 }
```

95.30.27 lib/unistd/getuid.c

Si veda la sezione [87.27](#).

```
5470001 #include <unistd.h>
5470002 #include <sys/types.h>
5470003 #include <sys/os32.h>
5470004 #include <errno.h>
5470005 //-----
5470006 uid_t
5470007 getuid (void)
5470008 {
5470009     sysmsg_uarea_t msg;
5470010     msg.path_cwd = NULL;
5470011     msg.path_cwd_size = 0;
5470012     sys (SYS_UAREA, &msg, (sizeof msg));
5470013     return (msg.uid);
5470014 }
```

95.30.28 lib/unistd/isatty.c

Si veda la sezione [88.69](#).

```
5480001 #include <sys/stat.h>
5480002 #include <sys/os32.h>
5480003 #include <unistd.h>
```

```
5480004 #include <sys/types.h>
5480005 #include <errno.h>
5480006 //-----
5480007 int
5480008 isatty (int fdn)
5480009 {
5480010     struct stat file_status;
5480011     //
5480012     // Verify to have valid input data.
5480013     //
5480014     if (fdn < 0)
5480015     {
5480016         errset (EBADF);
5480017         return (0);
5480018     }
5480019     //
5480020     // Verify the standard input.
5480021     //
5480022     if (fstat (fdn, &file_status) == 0)
5480023     {
5480024         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
5480025         {
5480026             return (1); // Meaning it is ok!
5480027         }
5480028         if (major (file_status.st_rdev) == DEV_TTY_MAJOR)
5480029         {
5480030             return (1); // Meaning it is ok!
5480031         }
5480032     }
5480033     else
5480034     {
5480035         errset (errno);
5480036         return (0);
5480037     }
5480038     //
5480039     // If here, it is not a terminal of any kind.
5480040     //
```

```
5480041     errset (EINVAL);
5480042     return (0);
5480043 }
```

95.30.29 lib/unistd/link.c



Si veda la sezione [87.30](#).

```
5490001 #include <unistd.h>
5490002 #include <string.h>
5490003 #include <sys/os32.h>
5490004 #include <errno.h>
5490005 #include <limits.h>
5490006 //-----
5490007 int
5490008 link (const char *path_old, const char *path_new)
5490009 {
5490010     sysmsg_link_t msg;
5490011     //
5490012     msg.path_old = path_old;
5490013     msg.path_new = path_new;
5490014     //
5490015     sys (SYS_LINK, &msg, (sizeof msg));
5490016     //
5490017     errno = msg.errno;
5490018     errln = msg.errln;
5490019     strncpy (errfn, msg.errfn, PATH_MAX);
5490020     return (msg.ret);
5490021 }
```

95.30.30 lib/unistd/lseek.c



Si veda la sezione [87.33](#).

```
5500001 #include <unistd.h>
5500002 #include <sys/types.h>
5500003 #include <sys/os32.h>
```

```
5500004 #include <errno.h>
5500005 #include <string.h>
5500006 //-----
5500007 off_t
5500008 lseek (int fdn, off_t offset, int whence)
5500009 {
5500010     sysmsg_lseek_t msg;
5500011     msg.fdn = fdn;
5500012     msg.offset = offset;
5500013     msg.whence = whence;
5500014     sys (SYS_LSEEK, &msg, (sizeof msg));
5500015     errno = msg.errno;
5500016     errln = msg.errln;
5500017     strncpy (errfn, msg.errfn, PATH_MAX);
5500018     return (msg.ret);
5500019 }
```

95.30.31 lib/unistd/pipe.c



Si veda la sezione [87.38](#).

```
5510001 #include <unistd.h>
5510002 #include <string.h>
5510003 #include <sys/os32.h>
5510004 #include <errno.h>
5510005 #include <limits.h>
5510006 //-----
5510007 int
5510008 pipe (int pipefd[2])
5510009 {
5510010     sysmsg_pipe_t msg;
5510011     //
5510012     if (pipefd == NULL)
5510013     {
5510014         errset (EINVAL);
5510015         return (-1);
5510016     }
```

```
5510017 //
5510018 sys (SYS_PIPE, &msg, (sizeof msg));
5510019 //
5510020 errno = msg.errno;
5510021 errln = msg.errln;
5510022 //
5510023 pipefd[0] = msg.pipefd[0];
5510024 pipefd[1] = msg.pipefd[1];
5510025 //
5510026 return (msg.ret);
5510027 }
```

95.30.32 lib/unistd/read.c



Si veda la sezione [87.39](#).

```
5520001 #include <unistd.h>
5520002 #include <sys/os32.h>
5520003 #include <errno.h>
5520004 #include <string.h>
5520005 #include <stdio.h>
5520006 #include <fcntl.h>
5520007 //-----
5520008 ssize_t
5520009 read (int fdn, void *buffer, size_t count)
5520010 {
5520011     sysmsg_read_t msg;
5520012     //
5520013     // Reduce size of read if necessary.
5520014     //
5520015     if (count > BUFSIZ)
5520016     {
5520017         count = BUFSIZ;
5520018     }
5520019     //
5520020     // Fill the message.
5520021     //
```

```
5520022 msg.fdn = fdn;
5520023 msg.buffer = buffer;
5520024 msg.count = count;
5520025 msg.fl_flags = 0;      // Not necessary.
5520026 msg.ret = 0;
5520027 //
5520028 // Repeat syscall, until something is received or
5520029 // end of file is
5520030 // reached.
5520031 //
5520032 while (1)
5520033 {
5520034     sys (SYS_READ, &msg, (sizeof msg));
5520035     if (msg.ret == 0)
5520036     {
5520037         //
5520038         // End of file.
5520039         //
5520040         break;
5520041     }
5520042     if (msg.ret < 0
5520043         && (msg.errno == EAGAIN
5520044             || msg.errno == EWOULDBLOCK))
5520045     {
5520046         //
5520047         // No data at the moment.
5520048         //
5520049         if (msg.fl_flags & O_NONBLOCK)
5520050         {
5520051             //
5520052             // Don't block.
5520053             //
5520054             break;
5520055         }
5520056         else
5520057         {
5520058             //
```



```
5520059          // Keep trying.
5520060          //
5520061          continue;
5520062      }
5520063  }
5520064  //
5520065  // Otherwise, we have read something.
5520066  //
5520067  break;
5520068  }
5520069  //
5520070  //
5520071  //
5520072  if (msg.ret < 0)
5520073  {
5520074      //
5520075      // No valid read.
5520076      //
5520077      errno = msg.errno;
5520078      errln = msg.errln;
5520079      strncpy (errfn, msg.errfn, PATH_MAX);
5520080      return (msg.ret);
5520081  }
5520082  //
5520083  if (msg.ret > count)
5520084  {
5520085      //
5520086      // A strange value was returned. Considering it
5520087      // a read error.
5520088      //
5520089      errset (EIO);      // I/O error.
5520090      return (-1);
5520091  }
5520092  //
5520093  // A valid read: return.
5520094  //
5520095  return (msg.ret);
```

5520096

}

95.30.33 lib/unistd/rmdir.c

<<

Si veda la sezione [87.41](#).

```
5530001 #include <unistd.h>
5530002 #include <string.h>
5530003 #include <sys/os32.h>
5530004 #include <errno.h>
5530005 #include <limits.h>
5530006 //-----
5530007 int
5530008 rmdir (const char *path)
5530009 {
5530010     sysmsg_stat_t msg_stat;
5530011     sysmsg_unlink_t msg_unlink;
5530012     //
5530013     msg_stat.path = path;
5530014     //
5530015     sys (SYS_STAT, &msg_stat, (sizeof msg_stat));
5530016     //
5530017     if (msg_stat.ret != 0)
5530018     {
5530019         errno = msg_stat.errno;
5530020         errln = msg_stat.errln;
5530021         strncpy (errfn, msg_stat.errfn, PATH_MAX);
5530022         return (msg_stat.ret);
5530023     }
5530024     //
5530025     if (!S_ISDIR (msg_stat.stat.st_mode))
5530026     {
5530027         errset (ENOTDIR); // Not a directory.
5530028         return (-1);
5530029     }
5530030     //
5530031     msg_unlink.path = path;
```

```
5530032 //
5530033 sys (SYS_UNLINK, &msg_unlink, (sizeof msg_unlink));
5530034 //
5530035 errno = msg_unlink.errno;
5530036 errln = msg_unlink.errln;
5530037 strncpy (errfn, msg_unlink.errfn, PATH_MAX);
5530038 return (msg_unlink.ret);
5530039 }
```

95.30.34 lib/unistd/sbrk.c



Si veda la sezione [87.5](#).

```
5540001 #include <unistd.h>
5540002 #include <string.h>
5540003 #include <sys/os32.h>
5540004 #include <errno.h>
5540005 #include <limits.h>
5540006 //-----
5540007 void *
5540008 sbrk (intptr_t increment)
5540009 {
5540010     sysmsg_sbrk_t msg_sbrk;
5540011     //
5540012     msg_sbrk.increment = increment;
5540013     //
5540014     sys (SYS_SBRK, &msg_sbrk, (sizeof msg_sbrk));
5540015     //
5540016     errno = msg_sbrk.errno;
5540017     errln = msg_sbrk.errln;
5540018     strncpy (errfn, msg_sbrk.errfn, PATH_MAX);
5540019     return (msg_sbrk.ret);
5540020 }
```

95.30.35 lib/unistd/setegid.c



Si veda la sezione [87.48](#).

```
5550001 #include <unistd.h>
5550002 #include <sys/types.h>
5550003 #include <sys/os32.h>
5550004 #include <errno.h>
5550005 #include <string.h>
5550006 //-----
5550007 int
5550008 setegid (gid_t gid)
5550009 {
5550010     sysmsg_setegid_t msg;
5550011     msg.ret = 0;
5550012     msg.errno = 0;
5550013     msg.egid = gid;
5550014     sys (SYS_SETEGID, &msg, (sizeof msg));
5550015     errno = msg.errno;
5550016     errln = msg.errln;
5550017     strncpy (errfn, msg.errfn, PATH_MAX);
5550018     return (msg.ret);
5550019 }
```

95.30.36 lib/unistd/seteuid.c



Si veda la sezione [87.51](#).

```
5560001 #include <unistd.h>
5560002 #include <sys/types.h>
5560003 #include <sys/os32.h>
5560004 #include <errno.h>
5560005 #include <string.h>
5560006 //-----
5560007 int
5560008 seteuid (uid_t uid)
5560009 {
5560010     sysmsg_seteuid_t msg;
```

```
5560011     msg.ret = 0;
5560012     msg.errno = 0;
5560013     msg.euid = uid;
5560014     sys (SYS_SETEGID, &msg, (sizeof msg));
5560015     errno = msg.errno;
5560016     errln = msg.errln;
5560017     strncpy (errfn, msg.errfn, PATH_MAX);
5560018     return (msg.ret);
5560019 }
```

95.30.37 lib/unistd/setgid.c

Si veda la sezione [87.48](#).

```
5570001 #include <unistd.h>
5570002 #include <sys/types.h>
5570003 #include <sys/os32.h>
5570004 #include <errno.h>
5570005 #include <string.h>
5570006 //-----
5570007 int
5570008 setgid (gid_t gid)
5570009 {
5570010     sysmsg_setgid_t msg;
5570011     msg.ret = 0;
5570012     msg.errno = 0;
5570013     msg.egid = gid;
5570014     sys (SYS_SETGID, &msg, (sizeof msg));
5570015     errno = msg.errno;
5570016     errln = msg.errln;
5570017     strncpy (errfn, msg.errfn, PATH_MAX);
5570018     return (msg.ret);
5570019 }
```

95.30.38 lib/unistd/setpgrp.c

<<

Si veda la sezione [87.50](#).

```
5580001 #include <unistd.h>
5580002 #include <sys/os32.h>
5580003 #include <stddef.h>
5580004 //-----
5580005 int
5580006 setpgrp (void)
5580007 {
5580008     sys (SYS_PGRP, NULL, (size_t) 0);
5580009     return (0);
5580010 }
```

95.30.39 lib/unistd/setuid.c

<<

Si veda la sezione [87.51](#).

```
5590001 #include <unistd.h>
5590002 #include <sys/types.h>
5590003 #include <sys/os32.h>
5590004 #include <errno.h>
5590005 #include <string.h>
5590006 //-----
5590007 int
5590008 setuid (uid_t uid)
5590009 {
5590010     sysmsg_setuid_t msg;
5590011     msg.ret = 0;
5590012     msg.errno = 0;
5590013     msg.euid = uid;
5590014     sys (SYS_SETUID, &msg, (sizeof msg));
5590015     errno = msg.errno;
5590016     errln = msg.errln;
5590017     strncpy (errfn, msg.errfn, PATH_MAX);
5590018     return (msg.ret);
5590019 }
```

95.30.40 lib/unistd/sleep.c



Si veda la sezione [87.53](#).

```
5600001 #include <unistd.h>
5600002 #include <sys/types.h>
5600003 #include <sys/os32.h>
5600004 #include <errno.h>
5600005 #include <time.h>
5600006 //-----
5600007 unsigned int
5600008 sleep (unsigned int seconds)
5600009 {
5600010     sysmsg_sleep_t msg;
5600011     time_t start;
5600012     time_t end;
5600013     int slept;
5600014     //
5600015     if (seconds == 0)
5600016     {
5600017         return (0);
5600018     }
5600019     //
5600020     msg.events = WAKEUP_EVENT_TIMER;
5600021     msg.seconds = seconds;
5600022     sys (SYS_SLEEP, &msg, (sizeof msg));
5600023     start = msg.ret;
5600024     end = time (NULL);
5600025     slept = end - msg.ret;
5600026     //
5600027     if (slept < 0)
5600028     {
5600029         return (seconds);
5600030     }
5600031     else if (slept < seconds)
5600032     {
5600033         return (seconds - slept);
5600034     }
```

```
5600035     else
5600036     {
5600037         return (0);
5600038     }
5600039 }
```

95.30.41 lib/unistd/ttyname.c

«

Si veda la sezione [88.133](#).

```
5610001 #include <sys/os32.h>
5610002 #include <sys/stat.h>
5610003 #include <unistd.h>
5610004 #include <sys/types.h>
5610005 #include <errno.h>
5610006 #include <limits.h>
5610007 //-----
5610008 char *
5610009 ttyname (int fdn)
5610010 {
5610011     dev_t dev_minor;
5610012     struct stat file_status;
5610013     static char name[PATH_MAX];
5610014     //
5610015     // Verify to have valid input data.
5610016     //
5610017     if (fdn < 0)
5610018     {
5610019         errset (EBADF);
5610020         return (NULL);
5610021     }
5610022     //
5610023     // Verify the file descriptor.
5610024     //
5610025     if (fstat (fdn, &file_status) == 0)
5610026     {
5610027         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
```



```
5610028     {
5610029         dev_minor = minor (file_status.st_rdev);
5610030         //
5610031         // If minor is equal to 0xFF, it is
5610032         // '/dev/console'.
5610033         //
5610034         if (dev_minor < 0xFF)
5610035             {
5610036                 sprintf (name, "/dev/console%i", dev_minor);
5610037             }
5610038         else
5610039             {
5610040                 strcpy (name, "/dev/console");
5610041             }
5610042         return (name);
5610043     }
5610044     else if (file_status.st_rdev == DEV_TTY)
5610045         {
5610046             strcpy (name, "/dev/tty");
5610047             return (name);
5610048         }
5610049     else
5610050         {
5610051             errset (ENOTTY);
5610052             return (NULL);
5610053         }
5610054     }
5610055     else
5610056         {
5610057             errset (errno);
5610058             return (NULL);
5610059         }
5610060 }
```

95.30.42 lib/unistd/unlink.c



Si veda la sezione [87.62](#).

```
5620001 #include <unistd.h>
5620002 #include <string.h>
5620003 #include <sys/os32.h>
5620004 #include <errno.h>
5620005 #include <limits.h>
5620006 //-----
5620007 int
5620008 unlink (const char *path)
5620009 {
5620010     sysmsg_unlink_t msg;
5620011     //
5620012     msg.path = path;
5620013     //
5620014     sys (SYS_UNLINK, &msg, (sizeof msg));
5620015     //
5620016     errno = msg.errno;
5620017     errln = msg.errln;
5620018     strncpy (errfn, msg.errfn, PATH_MAX);
5620019     return (msg.ret);
5620020 }
```

95.30.43 lib/unistd/write.c



Si veda la sezione [87.64](#).

```
5630001 #include <unistd.h>
5630002 #include <sys/os32.h>
5630003 #include <errno.h>
5630004 #include <string.h>
5630005 #include <stdio.h>
5630006 //-----
5630007 ssize_t
5630008 write (int fdn, const void *buffer, size_t count)
5630009 {
```

```
5630010     sysmsg_write_t msg;
5630011     //
5630012     // Reduce size of write if necessary.
5630013     //
5630014     if (count > BUFSIZ)
5630015     {
5630016         count = BUFSIZ;
5630017     }
5630018     //
5630019     // Fill the message.
5630020     //
5630021     msg.fdn = fdn;
5630022     msg.buffer = buffer;
5630023     msg.count = count;
5630024     //
5630025     // Syscall.
5630026     //
5630027     sys (SYS_WRITE, &msg, (sizeof msg));
5630028     //
5630029     // Check result and return.
5630030     //
5630031     if (msg.ret < 0)
5630032     {
5630033         //
5630034         // No valid write.
5630035         //
5630036         errno = msg.errno;
5630037         errln = msg.errln;
5630038         strncpy (errfn, msg.errfn, PATH_MAX);
5630039         return (msg.ret);
5630040     }
5630041     //
5630042     if (msg.ret > count)
5630043     {
5630044         //
5630045         // A strange value was returned. Considering it
5630046         // a read error.
```

```

5630047      //
5630048      errset (EIO);      // I/O error.
5630049      return (-1);
5630050    }
5630051    //
5630052    // A valid write return.
5630053    //
5630054    return (msg.ret);
5630055  }

```

95.31 os32: «lib/utime.h»

«

Si veda la sezione 91.3.

```

5640001  #ifndef _UTIME_H
5640002  #define _UTIME_H          1
5640003  //-----
5640004  #include <restrict.h>
5640005  #include <sys/types.h>  // time_t
5640006  //-----
5640007  struct utimbuf
5640008  {
5640009      time_t actime;
5640010      time_t modtime;
5640011  };
5640012  //-----
5640013  int utime (const char *path, const struct utimbuf *times);
5640014  //-----
5640015
5640016  #endif

```

95.31.1 lib/utime/utime.c 2229

95.31.1 lib/utime/utime.c



Si veda la sezione [91.3](#).

```
5650001 #include <utime.h>
5650002 #include <errno.h>
5650003 //-----
5650004 int
5650005 utime (const char *path, const struct utimbuf *times)
5650006 {
5650007     //
5650008     // Currently not implemented.
5650009     //
5650010     return (0);
5650011 }
```


Sorgenti delle applicazioni



96.1	os32: directory «applic/»	2233
96.1.1	applic/MAKEDEV.c	2233
96.1.2	applic/aaa.c	2238
96.1.3	applic/allocated.c	2239
96.1.4	applic/arp.c	2241
96.1.5	applic/bbb.c	2243
96.1.6	applic/cat.c	2244
96.1.7	applic/ccc.c	2246
96.1.8	applic/chgrp.c	2247
96.1.9	applic/chmod.c	2250
96.1.10	applic/chown.c	2252
96.1.11	applic/cp.c	2255
96.1.12	applic/crt0.mer.s	2261
96.1.13	applic/crt0.sep.s	2265
96.1.14	applic/date.c	2269
96.1.15	applic/ed.c	2273
96.1.16	applic/getty.c	2314
96.1.17	applic/http.c	2317
96.1.18	applic/init.c	2333
96.1.19	applic/ipconfig.c	2340
96.1.20	applic/kill.c	2343
96.1.21	applic/ln.c	2349
96.1.22	applic/login.c	2353

96.1.23	applic/ls.c	2358
96.1.24	applic/man.c	2369
96.1.25	applic/mkdir.c	2378
96.1.26	applic/mmcheck.c	2383
96.1.27	applic/more.c	2390
96.1.28	applic/mount.c	2397
96.1.29	applic/nc.c	2399
96.1.30	applic/ping.c	2409
96.1.31	applic/ps.c	2414
96.1.32	applic/rm.c	2422
96.1.33	applic/rmdir.c	2424
96.1.34	applic/route.c	2426
96.1.35	applic/shell.c	2429
96.1.36	applic/t_fcntl.c	2436
96.1.37	applic/t_fifo.c	2437
96.1.38	applic/t_grp.c	2440
96.1.39	applic/t_nc.c	2441
96.1.40	applic/t_ping2.c	2451
96.1.41	applic/t_pipe.c	2453
96.1.42	applic/t_read.c	2456
96.1.43	applic/t_ret.c	2458
96.1.44	applic/t_rx_udp.c	2458
96.1.45	applic/t_scr.c	2461
96.1.46	applic/t_setjmp.c	2462
96.1.47	applic/t_sig.c	2463

96.1.48	applic/t_sig2.c	2464
96.1.49	applic/t_tx_tcp.c	2466
96.1.50	applic/t_tx_udp.c	2469
96.1.51	applic/touch.c	2473
96.1.52	applic/tty.c	2475
96.1.53	applic/umount.c	2477
96.1.54	applic/yes.c	2478

96.1 os32: directory «applic/»

«

96.1.1 applic/MAKEDEV.c

«

Si veda la sezione [92.6](#).

```
5660001 #include <unistd.h>
5660002 #include <stdlib.h>
5660003 #include <sys/stat.h>
5660004 #include <fcntl.h>
5660005 #include <kernel/dev.h>
5660006 #include <stdio.h>
5660007 //-----
5660008 int
5660009 main (void)
5660010 {
5660011     int status;
5660012     //
5660013     status =
5660014         mknod ("mem", (mode_t) (S_IFCHR | 0444),
5660015             (dev_t) DEV_MEM);
5660016     if (status)
5660017         perror (NULL);
5660018     status =
5660019         mknod ("null", (mode_t) (S_IFCHR | 0666),
5660020             (dev_t) DEV_NULL);
```

```
5660021     if (status)
5660022         perror (NULL);
5660023     status =
5660024         mknod ("port", (mode_t) (S_IFCHR | 0644),
5660025             (dev_t) DEV_PORT);
5660026     if (status)
5660027         perror (NULL);
5660028     status =
5660029         mknod ("zero", (mode_t) (S_IFCHR | 0666),
5660030             (dev_t) DEV_ZERO);
5660031     if (status)
5660032         perror (NULL);
5660033     status =
5660034         mknod ("tty", (mode_t) (S_IFCHR | 0666),
5660035             (dev_t) DEV_TTY);
5660036     if (status)
5660037         perror (NULL);
5660038     status = mknod ("kmem_ps", (mode_t) (S_IFCHR | 0444),
5660039                 (dev_t) DEV_KMEM_PS);
5660040     if (status)
5660041         perror (NULL);
5660042     status =
5660043         mknod ("kmem_mmp", (mode_t) (S_IFCHR | 0444),
5660044             (dev_t) DEV_KMEM_MMP);
5660045     if (status)
5660046         perror (NULL);
5660047     status = mknod ("kmem_sb", (mode_t) (S_IFCHR | 0444),
5660048                 (dev_t) DEV_KMEM_SB);
5660049     if (status)
5660050         perror (NULL);
5660051     status =
5660052         mknod ("kmem_inode", (mode_t) (S_IFCHR | 0444),
5660053             (dev_t) DEV_KMEM_INODE);
5660054     if (status)
5660055         perror (NULL);
5660056     status =
5660057         mknod ("kmem_file", (mode_t) (S_IFCHR | 0444),
```

```
5660058         (dev_t) DEV_KMEM_FILE);
5660059     if (status)
5660060         perror (NULL);
5660061     status = mknod ("console", (mode_t) (S_IFCHR | 0644),
5660062                 (dev_t) DEV_CONSOLE);
5660063     if (status)
5660064         perror (NULL);
5660065     status =
5660066         mknod ("console0", (mode_t) (S_IFCHR | 0644),
5660067             (dev_t) DEV_CONSOLE0);
5660068     if (status)
5660069         perror (NULL);
5660070     status =
5660071         mknod ("console1", (mode_t) (S_IFCHR | 0644),
5660072             (dev_t) DEV_CONSOLE1);
5660073     if (status)
5660074         perror (NULL);
5660075     status =
5660076         mknod ("console2", (mode_t) (S_IFCHR | 0644),
5660077             (dev_t) DEV_CONSOLE2);
5660078     if (status)
5660079         perror (NULL);
5660080     status =
5660081         mknod ("console3", (mode_t) (S_IFCHR | 0644),
5660082             (dev_t) DEV_CONSOLE3);
5660083     if (status)
5660084         perror (NULL);
5660085
5660086     status =
5660087         mknod ("dm00", (mode_t) (S_IFBLK | 0644),
5660088             (dev_t) DEV_DM00);
5660089     if (status)
5660090         perror (NULL);
5660091     status =
5660092         mknod ("dm01", (mode_t) (S_IFBLK | 0644),
5660093             (dev_t) DEV_DM01);
5660094     if (status)
```

```
5660095     perror (NULL);
5660096     status =
5660097     mknod ("dm02", (mode_t) (S_IFBLK | 0644),
5660098           (dev_t) DEV_DM02);
5660099     if (status)
5660100         perror (NULL);
5660101     status =
5660102     mknod ("dm03", (mode_t) (S_IFBLK | 0644),
5660103           (dev_t) DEV_DM03);
5660104     if (status)
5660105         perror (NULL);
5660106     status =
5660107     mknod ("dm04", (mode_t) (S_IFBLK | 0644),
5660108           (dev_t) DEV_DM04);
5660109     if (status)
5660110         perror (NULL);
5660111     status =
5660112     mknod ("dm10", (mode_t) (S_IFBLK | 0644),
5660113           (dev_t) DEV_DM10);
5660114     if (status)
5660115         perror (NULL);
5660116     status =
5660117     mknod ("dm11", (mode_t) (S_IFBLK | 0644),
5660118           (dev_t) DEV_DM11);
5660119     if (status)
5660120         perror (NULL);
5660121     status =
5660122     mknod ("dm12", (mode_t) (S_IFBLK | 0644),
5660123           (dev_t) DEV_DM12);
5660124     if (status)
5660125         perror (NULL);
5660126     status =
5660127     mknod ("dm13", (mode_t) (S_IFBLK | 0644),
5660128           (dev_t) DEV_DM13);
5660129     if (status)
5660130         perror (NULL);
5660131     status =
```

```
5660132     mknod ("dm14", (mode_t) (S_IFBLK | 0644),
5660133         (dev_t) DEV_DM14);
5660134     if (status)
5660135         perror (NULL);
5660136     status =
5660137         mknod ("dm20", (mode_t) (S_IFBLK | 0644),
5660138             (dev_t) DEV_DM20);
5660139     if (status)
5660140         perror (NULL);
5660141     status =
5660142         mknod ("dm21", (mode_t) (S_IFBLK | 0644),
5660143             (dev_t) DEV_DM21);
5660144     if (status)
5660145         perror (NULL);
5660146     status =
5660147         mknod ("dm22", (mode_t) (S_IFBLK | 0644),
5660148             (dev_t) DEV_DM22);
5660149     if (status)
5660150         perror (NULL);
5660151     status =
5660152         mknod ("dm23", (mode_t) (S_IFBLK | 0644),
5660153             (dev_t) DEV_DM23);
5660154     if (status)
5660155         perror (NULL);
5660156     status =
5660157         mknod ("dm24", (mode_t) (S_IFBLK | 0644),
5660158             (dev_t) DEV_DM24);
5660159     if (status)
5660160         perror (NULL);
5660161     status =
5660162         mknod ("dm30", (mode_t) (S_IFBLK | 0644),
5660163             (dev_t) DEV_DM30);
5660164     if (status)
5660165         perror (NULL);
5660166     status =
5660167         mknod ("dm31", (mode_t) (S_IFBLK | 0644),
5660168             (dev_t) DEV_DM31);
```

```
5660169     if (status)
5660170         perror (NULL);
5660171     status =
5660172         mknod ("dm32", (mode_t) (S_IFBLK | 0644),
5660173             (dev_t) DEV_DM32);
5660174     if (status)
5660175         perror (NULL);
5660176     status =
5660177         mknod ("dm33", (mode_t) (S_IFBLK | 0644),
5660178             (dev_t) DEV_DM33);
5660179     if (status)
5660180         perror (NULL);
5660181     status =
5660182         mknod ("dm34", (mode_t) (S_IFBLK | 0644),
5660183             (dev_t) DEV_DM34);
5660184     if (status)
5660185         perror (NULL);
5660186     //
5660187     return (0);
5660188 }
```

96.1.2 applic/aaa.c



Si veda la sezione [86.1](#).

```
5670001 #include <unistd.h>
5670002 #include <stdio.h>
5670003 //-----
5670004 int
5670005 main (void)
5670006 {
5670007     unsigned int count;
5670008     for (count = 0; count < 60; count++)
5670009     {
5670010         printf ("a");
5670011         sleep (1);
5670012     }
```

```
5670013     return (8);
5670014 }
```

96.1.3 applic/allocated.c

Si veda la sezione [86.2](#).

```
5680001 #include <sys/os32.h>
5680002 #include <kernel/memory.h>
5680003 #include <unistd.h>
5680004 #include <stdio.h>
5680005 #include <fcntl.h>
5680006 #include <unistd.h>
5680007 #include <stdlib.h>
5680008 //-----
5680009 uint32_t mb_table[MEM_MAX_BLOCKS / 32]; // Memory
5680010                                           // blocks map.
5680011 unsigned int mb_max = MEM_MAX_BLOCKS; // Memory
5680012                                           // blocks max.
5680013 //-----
5680014 int
5680015 main (int argc, char *argv[], char *envp[])
5680016 {
5680017     unsigned int block;
5680018     unsigned int blocks = MEM_MAX_BLOCKS;
5680019     int i;
5680020     int j;
5680021     uint32_t mask;
5680022     unsigned int start = 0;
5680023     unsigned int stop = 0;
5680024     unsigned int status = 0;
5680025     int fd;
5680026     ssize_t size_read;
5680027     char *buffer = (char *) mb_table;
5680028     //
5680029     fd = open ("/dev/kmem_map", O_RDONLY);
5680030     if (fd < 0)
```

```
5680031     {
5680032         printf ("%s] Cannot open \"/dev/kmem_map\" ",
5680033                 argv[0]);
5680034         perror (NULL);
5680035         return (0);
5680036     }
5680037     //
5680038     lseek (fd, (off_t) 0, SEEK_SET);
5680039     for (i = 0; i < (MEM_MAX_BLOCKS / 8); i += size_read)
5680040     {
5680041         size_read = read (fd, &buffer[i], BUFSIZ);
5680042         if (size_read < 0)
5680043         {
5680044             printf
5680045                 ("%s] Cannot read "
5680046                  "\"/dev/kmem_map\" %i %i ",
5680047                 argv[0], size_read, sizeof (mb_table));
5680048             perror (NULL);
5680049             return (0);
5680050         }
5680051     }
5680052     //
5680053     printf ("Hex mem map, blocks of %x:", MEM_BLOCK_SIZE);
5680054     //
5680055     for (block = 0; block < blocks; block++)
5680056     {
5680057         i = block / 32;
5680058         j = block % 32;
5680059         mask = 0x80000000 >> j;
5680060         if (mb_table[i] & mask)
5680061         {
5680062             //
5680063             // Allocated block
5680064             //
5680065             if (status == 0)
5680066             {
5680067                 status = 1;
```



```
5680068         start = block;
5680069     }
5680070 }
5680071 else
5680072 {
5680073     //
5680074     // Not allocated block.
5680075     //
5680076     if (status == 1)
5680077     {
5680078         status = 0;
5680079         stop = block;
5680080     }
5680081 }
5680082 //
5680083 //
5680084 //
5680085 if (stop > 0)
5680086 {
5680087     printf (" %x-%x", start, stop);
5680088     start = 0;
5680089     stop = 0;
5680090 }
5680091 }
5680092 printf ("\n");
5680093
5680094 return (0);
5680095 }
```

96.1.4 applic/arp.c

Si veda la sezione [92.1](#).

```
5690001 #include <sys/os32.h>
5690002 #include <kernel/net/arp.h>
5690003 #include <unistd.h>
5690004 #include <stdio.h>
```

```
5690005 #include <fcntl.h>
5690006 #include <unistd.h>
5690007 #include <stdlib.h>
5690008 #include <time.h>
5690009 //-----
5690010 int
5690011 main (int argc, char *argv[], char *envp[])
5690012 {
5690013     int fd;
5690014     ssize_t size_read;
5690015     char buffer[sizeof (arp_t)];
5690016     int a;
5690017     arp_t *arp_table_item;
5690018     //
5690019     // All options are ignored.
5690020     //
5690021     // Open '/dev/kmem_arp', to get the ARP table.
5690022     //
5690023     fd = open ("/dev/kmem_arp", O_RDONLY);
5690024     if (fd < 0)
5690025     {
5690026         printf ("%s] Cannot open \"/dev/kmem_arp\" ",
5690027                 argv[0]);
5690028         perror (NULL);
5690029         exit (0);
5690030     }
5690031     //
5690032     // Scan ARP items and then print body.
5690033     //
5690034     for (a = 0; a < ARP_MAX_ITEMS; a++)
5690035     {
5690036         lseek (fd, (off_t) a, SEEK_SET);
5690037         size_read = read (fd, buffer, sizeof (arp_t));
5690038         if (size_read < sizeof (arp_t))
5690039         {
5690040             printf
5690041                 ("%s] Cannot read "
```

```

5690042         "\dev/kmem_arp\" item %i ", argv[0], a);
5690043     perror (NULL);
5690044     continue;
5690045 }
5690046 arp_table_item = (arp_t *) buffer;
5690047 if (arp_table_item->time > 0)
5690048 {
5690049     printf ("%i.%i.%i.%i  ",
5690050             arp_table_item->ip >> 24 & 0x000000FF,
5690051             arp_table_item->ip >> 16 & 0x000000FF,
5690052             arp_table_item->ip >> 8 & 0x000000FF,
5690053             arp_table_item->ip >> 0 & 0x000000FF);
5690054     //
5690055     printf ("%02x:%02x:%02x:%02x:%02x:%02x  ",
5690056             arp_table_item->mac[0],
5690057             arp_table_item->mac[1],
5690058             arp_table_item->mac[2],
5690059             arp_table_item->mac[3],
5690060             arp_table_item->mac[4],
5690061             arp_table_item->mac[5]);
5690062     //
5690063     printf ("%3us\n",
5690064             (unsigned int) (time (NULL) -
5690065                             arp_table_item->time));
5690066 }
5690067 }
5690068 close (fd);
5690069 return (0);
5690070 }

```

96.1.5 applic/bbb.c

Si veda la sezione [86.1](#).

```

5700001 #include <unistd.h>
5700002 #include <stdio.h>
5700003 #include <stdlib.h>

```



```
5700004 //-----  
5700005 int  
5700006 main (void)  
5700007 {  
5700008     unsigned int count;  
5700009     for (count = 0; count < 30; count++)  
5700010     {  
5700011         printf ("b");  
5700012         sleep (2);  
5700013     }  
5700014     exit (0);  
5700015     return (0);  
5700016 }
```

96.1.6 applic/cat.c



Si veda la sezione [86.4](#).

```
5710001 #include <fcntl.h>  
5710002 #include <sys/stat.h>  
5710003 #include <stddef.h>  
5710004 #include <unistd.h>  
5710005 #include <stdio.h>  
5710006 #include <stdlib.h>  
5710007 #include <errno.h>  
5710008 //-----  
5710009 static void cat_file_descriptor (int fd);  
5710010 //-----  
5710011 int  
5710012 main (int argc, char *argv[], char *envp[])  
5710013 {  
5710014     int i;  
5710015     int fd;  
5710016     struct stat file_status;  
5710017     //  
5710018     // Check if the input comes from standard input.  
5710019     //
```

```
5710020     if (argc < 2)
5710021     {
5710022         cat_file_descriptor (STDIN_FILENO);
5710023         exit (0);
5710024     }
5710025     //
5710026     // There is at least an argument: scan them.
5710027     //
5710028     for (i = 1; i < argc; i++)
5710029     {
5710030         //
5710031         // Verify if the file exists.
5710032         //
5710033         if (stat (argv[i], &file_status) != 0)
5710034         {
5710035             fprintf (stderr,
5710036                 "File \"%s\" does not exist!\n",
5710037                 argv[i]);
5710038             continue;
5710039         }
5710040         //
5710041         // File exists: check the file type.
5710042         //
5710043         if (S_ISDIR (file_status.st_mode))
5710044         {
5710045             fprintf (stderr, "Cannot \"cat\" "
5710046                 "\"%s\": it is a directory!\n", argv[i]);
5710047             continue;
5710048         }
5710049         //
5710050         // File exists and can be "cat"ed.
5710051         //
5710052         fd = open (argv[i], O_RDONLY);
5710053         if (fd >= 0)
5710054         {
5710055             cat_file_descriptor (fd);
5710056             close (fd);
```

```
5710057     }
5710058     else
5710059     {
5710060         perror (NULL);
5710061         exit (1);
5710062     }
5710063 }
5710064 return (0);
5710065 }
5710066
5710067 //-----
5710068 static void
5710069 cat_file_descriptor (int fd)
5710070 {
5710071     ssize_t count;
5710072     char buffer[BUFSIZ];
5710073
5710074     for (;;)
5710075     {
5710076         count = read (fd, buffer, (size_t) BUFSIZ);
5710077         if (count > 0)
5710078         {
5710079             write (STDOUT_FILENO, buffer, (size_t) count);
5710080         }
5710081         else
5710082         {
5710083             break;
5710084         }
5710085     }
5710086 }
```

96.1.7 applic/ccc.c



Si veda la sezione [86.1](#).

```
5720001 #include <unistd.h>
5720002 #include <stdlib.h>
```

```
5720003 #include <signal.h>
5720004 //-----
5720005 int
5720006 main (void)
5720007 {
5720008     pid_t pid;
5720009     // -----
5720010     pid = fork ();
5720011     if (pid == 0)
5720012     {
5720013         setuid ((uid_t) 10);
5720014         execve ("/bin/aaa", NULL, NULL);
5720015         exit (0);
5720016     }
5720017     // -----
5720018     pid = fork ();
5720019     if (pid == 0)
5720020     {
5720021         setuid ((uid_t) 11);
5720022         execve ("/bin/bbb", NULL, NULL);
5720023         exit (0);
5720024     }
5720025     // -----
5720026     while (1)
5720027     {
5720028         ; // Just loop, to consume CPU time: it must be
5720029         // killed manually.
5720030     }
5720031     return (0);
5720032 }
```

96.1.8 applic/chgrp.c

Si veda la sezione [86.6](#).

```
5730001 #include <unistd.h>
5730002 #include <stdlib.h>
```

```
5730003 #include <sys/stat.h>
5730004 #include <sys/types.h>
5730005 #include <fcntl.h>
5730006 #include <errno.h>
5730007 #include <stdio.h>
5730008 #include <ctype.h>
5730009 #include <grp.h>
5730010 //-----
5730011 static void usage (void);
5730012 //-----
5730013 int
5730014 main (int argc, char *argv[], char *envp[])
5730015 {
5730016     char *group;
5730017     int gid;
5730018     struct group *grs;
5730019     struct stat file_status;
5730020     int a;          // Argument index.
5730021     int status;
5730022     //
5730023     //
5730024     //
5730025     if (argc < 3)
5730026     {
5730027         usage ();
5730028         return (1);
5730029     }
5730030     //
5730031     // Get group id number.
5730032     //
5730033     group = argv[1];
5730034     if (isdigit (*group))
5730035     {
5730036         gid = atoi (group);
5730037     }
5730038     else
5730039     {
```



```
5730040     grs = getgrnam (group);
5730041     if (grs == NULL)
5730042     {
5730043         fprintf (stderr, "Unknown group \"%s\"!\n",
5730044                 group);
5730045         return (2);
5730046     }
5730047     gid = grs->gr_gid;
5730048 }
5730049 //
5730050 // Now we have the group id. Start scanning file
5730051 // names.
5730052 //
5730053 for (a = 2; a < argc; a++)
5730054 {
5730055     //
5730056     // Verify if the file exists, through the return
5730057     // value of
5730058     // 'stat()'. No other checks are made.
5730059     //
5730060     if (stat (argv[a], &file_status) == 0)
5730061     {
5730062         //
5730063         // Try to change ownership.
5730064         //
5730065         status = chown (argv[a], file_status.st_uid, gid);
5730066         if (status != 0)
5730067         {
5730068             perror (NULL);
5730069             return (3);
5730070         }
5730071     }
5730072     else
5730073     {
5730074         fprintf (stderr,
5730075                 "File \"%s\" does not exist!\n",
5730076                 argv[a]);
```

```
5730077         continue;
5730078     }
5730079 }
5730080 //
5730081 // All done.
5730082 //
5730083 return (0);
5730084 }
5730085
5730086 //-----
5730087 static void
5730088 usage (void)
5730089 {
5730090     fprintf (stderr, "Usage:  chown GROUP|UID FILE...\n");
5730091     fprintf (stderr, "Example: chown group my_file\n");
5730092 }
```

96.1.9 applic/chmod.c



Si veda la sezione [86.7](#).

```
5740001 #include <unistd.h>
5740002 #include <stdlib.h>
5740003 #include <sys/stat.h>
5740004 #include <sys/types.h>
5740005 #include <fcntl.h>
5740006 #include <errno.h>
5740007 #include <signal.h>
5740008 #include <stdio.h>
5740009 #include <sys/wait.h>
5740010 #include <stdio.h>
5740011 #include <string.h>
5740012 #include <limits.h>
5740013 #include <sys/os32.h>
5740014 //-----
5740015 static void usage (void);
5740016 //-----
```

```
5740017 int
5740018 main (int argc, char *argv[], char *envp[])
5740019 {
5740020     int status;
5740021     mode_t mode;
5740022     char *m;      // Pointer inside the octal mode
5740023     // string.
5740024     int digit;
5740025     int a;      // Argument index.
5740026     //
5740027     //
5740028     //
5740029     if (argc < 3)
5740030     {
5740031         usage ();
5740032         return (1);
5740033     }
5740034     //
5740035     // Get mode: must be the first argument.
5740036     //
5740037     for (m = argv[1]; *m != 0; m++)
5740038     {
5740039         digit = (*m - '0');
5740040         if (digit < 0 || digit > 7)
5740041         {
5740042             usage ();
5740043             return (2);
5740044         }
5740045         mode = mode * 8 + digit;
5740046     }
5740047     //
5740048     // System call for all the remaining arguments.
5740049     //
5740050     for (a = 2; a < argc; a++)
5740051     {
5740052         status = chmod (argv[a], mode);
5740053         if (status != 0)
```

```
5740054         {
5740055             perror (argv[a]);
5740056             return (3);
5740057         }
5740058     }
5740059     //
5740060     // All done.
5740061     //
5740062     return (0);
5740063 }
5740064
5740065 //-----
5740066 static void
5740067 usage (void)
5740068 {
5740069     fprintf (stderr, "Usage:  chmod OCTAL_MODE FILE...\n");
5740070     fprintf (stderr, "Example: chmod 0640 my_file\n");
5740071 }
```

96.1.10 applic/chown.c



Si veda la sezione [86.8](#).

```
5750001 #include <unistd.h>
5750002 #include <stdlib.h>
5750003 #include <sys/stat.h>
5750004 #include <sys/types.h>
5750005 #include <fcntl.h>
5750006 #include <errno.h>
5750007 #include <stdio.h>
5750008 #include <ctype.h>
5750009 #include <pwd.h>
5750010 //-----
5750011 static void usage (void);
5750012 //-----
5750013 int
5750014 main (int argc, char *argv[], char *envp[])
```

```
5750015 {
5750016     char *user;
5750017     int uid;
5750018     struct passwd *pws;
5750019     struct stat file_status;
5750020     int a;          // Argument index.
5750021     int status;
5750022     //
5750023     //
5750024     //
5750025     if (argc < 3)
5750026     {
5750027         usage ();
5750028         return (1);
5750029     }
5750030     //
5750031     // Get user id number.
5750032     //
5750033     user = argv[1];
5750034     if (isdigit (*user))
5750035     {
5750036         uid = atoi (user);
5750037     }
5750038     else
5750039     {
5750040         pws = getpwnam (user);
5750041         if (pws == NULL)
5750042         {
5750043             fprintf (stderr, "Unknown user \"%s\"!\n", user);
5750044             return (2);
5750045         }
5750046         uid = pws->pw_uid;
5750047     }
5750048     //
5750049     // Now we have the user id. Start scanning file
5750050     // names.
5750051     //
```

```
5750052     for (a = 2; a < argc; a++)
5750053     {
5750054         //
5750055         // Verify if the file exists, through the return
5750056         // value of
5750057         // 'stat()'. No other checks are made.
5750058         //
5750059         if (stat (argv[a], &file_status) == 0)
5750060         {
5750061             //
5750062             // Try to change ownership.
5750063             //
5750064             status = chown (argv[a], uid, file_status.st_gid);
5750065             if (status != 0)
5750066             {
5750067                 perror (NULL);
5750068                 return (3);
5750069             }
5750070         }
5750071         else
5750072         {
5750073             fprintf (stderr,
5750074                     "File \"%s\" does not exist!\n",
5750075                     argv[a]);
5750076             continue;
5750077         }
5750078     }
5750079     //
5750080     // All done.
5750081     //
5750082     return (0);
5750083 }
5750084
5750085 //-----
5750086 static void
5750087 usage (void)
5750088 {
```

```
5750089     fprintf (stderr, "Usage:  chown USER|UID FILE...\n");
5750090     fprintf (stderr, "Example: chown user my_file\n");
5750091 }
```

96.1.11 applic/cp.c

Si veda la sezione [86.9](#).

```
5760001 #include <sys/os32.h>
5760002 #include <sys/stat.h>
5760003 #include <sys/types.h>
5760004 #include <unistd.h>
5760005 #include <stdlib.h>
5760006 #include <fcntl.h>
5760007 #include <errno.h>
5760008 #include <signal.h>
5760009 #include <stdio.h>
5760010 #include <string.h>
5760011 #include <limits.h>
5760012 #include <libgen.h>
5760013 //-----
5760014 static void usage (void);
5760015 //-----
5760016 int
5760017 main (int argc, char *argv[], char *envp[])
5760018 {
5760019     char *source;
5760020     char *destination;
5760021     char *destination_full;
5760022     struct stat file_status;
5760023     int dest_is_a_dir = 0;
5760024     int a;           // Argument index.
5760025     char path[PATH_MAX];
5760026     int fd_source = -1;
5760027     int fd_destination = -1;
5760028     char buffer_in[BUFSIZ];
5760029     char *buffer_out;
```

```
5760030     ssize_t count_in;      // Read counter.
5760031     ssize_t count_out;   // Write counter.
5760032     //
5760033     // There must be at least two arguments, plus the
5760034     // program name.
5760035     //
5760036     if (argc < 3)
5760037     {
5760038         usage ();
5760039         return (1);
5760040     }
5760041     //
5760042     // Select the last argument as the destination.
5760043     //
5760044     destination = argv[argc - 1];
5760045     //
5760046     // Check if it is a directory and save it in a flag.
5760047     //
5760048     if (stat (destination, &file_status) == 0)
5760049     {
5760050         if (S_ISDIR (file_status.st_mode))
5760051         {
5760052             dest_is_a_dir = 1;
5760053         }
5760054     }
5760055     //
5760056     // If there are more than two arguments, verify that
5760057     // the last
5760058     // one is a directory.
5760059     //
5760060     if (argc > 3)
5760061     {
5760062         if (!dest_is_a_dir)
5760063         {
5760064             usage ();
5760065             fprintf (stderr, "The destination \"%s\" ",
5760066                     destination);
```



```
5760067         fprintf (stderr, "is not a directory!\n");
5760068         return (1);
5760069     }
5760070 }
5760071 //
5760072 // Scan the arguments, excluded the last, that is
5760073 // the destination.
5760074 //
5760075 for (a = 1; a < (argc - 1); a++)
5760076 {
5760077     //
5760078     // Source.
5760079     //
5760080     source = argv[a];
5760081     //
5760082     // Verify access permissions.
5760083     //
5760084     if (access (source, R_OK) < 0)
5760085     {
5760086         perror (source);
5760087         continue;
5760088     }
5760089     //
5760090     // Destination.
5760091     //
5760092     // If it is a directory, the destination path
5760093     // must be corrected.
5760094     //
5760095     if (dest_is_a_dir)
5760096     {
5760097         path[0] = 0;
5760098         strcat (path, destination);
5760099         strcat (path, "/");
5760100         strcat (path, basename (source));
5760101         //
5760102         // Update the destination path.
5760103         //
```

```
5760104     destination_full = path;
5760105     }
5760106     else
5760107     {
5760108         destination_full = destination;
5760109     }
5760110     //
5760111     // Check if destination file exists.
5760112     //
5760113     if (stat (destination_full, &file_status) == 0)
5760114     {
5760115         fprintf (stderr,
5760116                 "The destination file, \"%s\", ",
5760117                 destination_full);
5760118         fprintf (stderr, "already exists!\n");
5760119         continue;
5760120     }
5760121     //
5760122     // Everything is ready for the copy.
5760123     //
5760124     fd_source = open (source, O_RDONLY);
5760125     if (fd_source < 0)
5760126     {
5760127         perror (source);
5760128         //
5760129         // Continue with the next file.
5760130         //
5760131         continue;
5760132     }
5760133     //
5760134     fd_destination = creat (destination_full, 0777);
5760135     if (fd_destination < 0)
5760136     {
5760137         perror (destination);
5760138         close (fd_source);
5760139         //
5760140         // Continue with the next file.
```

```
5760141         //
5760142         continue;
5760143     }
5760144     //
5760145     // Copy the data.
5760146     //
5760147     while (1)
5760148     {
5760149         count_in =
5760150             read (fd_source, buffer_in, (size_t) BUFSIZ);
5760151         if (count_in > 0)
5760152         {
5760153             for (buffer_out = buffer_in; count_in > 0;)
5760154                 {
5760155                     count_out =
5760156                         write (fd_destination, buffer_out,
5760157                             (size_t) count_in);
5760158                     if (count_out < 0)
5760159                         {
5760160                             perror (destination);
5760161                             close (fd_source);
5760162                             close (fd_destination);
5760163                             return (3);
5760164                         }
5760165                     //
5760166                     // If not all data is written,
5760167                     // continue writing,
5760168                     // but change the buffer start
5760169                     // position and the
5760170                     // amount to be written.
5760171                     //
5760172                     buffer_out += count_out;
5760173                     count_in -= count_out;
5760174                 }
5760175             }
5760176         else if (count_in < 0)
5760177         {
```

```
5760178         perror (source);
5760179         close (fd_source);
5760180         close (fd_destination);
5760181     }
5760182     else
5760183     {
5760184         break;
5760185     }
5760186 }
5760187 //
5760188 if (close (fd_source))
5760189 {
5760190     perror (source);
5760191 }
5760192 if (close (fd_destination))
5760193 {
5760194     perror (destination);
5760195     return (4);
5760196 }
5760197 }
5760198 //
5760199 // All done.
5760200 //
5760201 return (0);
5760202 }
5760203
5760204 //-----
5760205 static void
5760206 usage (void)
5760207 {
5760208     fprintf (stderr, "Usage: cp OLD_NAME NEW_NAME\n");
5760209     fprintf (stderr, "          cp FILE... DIRECTORY\n");
5760210 }
```

96.1.12 applic/crt0.mer.s



Si veda la sezione [84.5.2](#).

```
5770001 .extern main
5770002 .extern _stdio_stream_setup
5770003 .extern _dirent_directory_stream_setup
5770004 .extern _atexit_setup
5770005 .extern _environment_setup
5770006 .global startup
5770007 .global _data_end
5770008 #-----
5770009 # Please note that, all segment descriptors are already
5770010 # set from the scheduler, and there is also data inside
5770011 # the stack, so that the call to 'main()' function will
5770012 # result as expected.
5770013 #-----
5770014 # The following statement says that the code will start
5770015 # at "startup" label.
5770016 #-----
5770017 .section .text
5770018 #-----
5770019 startup:
5770020     #
5770021     # Jump after initial data.
5770022     #
5770023     jmp startup_code
5770024     #
5770025 filler:
5770026     #
5770027     # After four bytes, from the start, there is the
5770028     # magic number and other data.
5770029     #
5770030     .space (0x0004 - (filler - startup))
5770031     #
5770032 magic:
5770033     .quad 0x6F7333326170706C    # os32appl
5770034     ;
```

```
5770035 doffset: #
5770036     .int _text_start # Data offset from start.
5770037 etext: #
5770038     .int _text_end # End of code
5770039 edata: #
5770040     .int _data_end # End of initialized data.
5770041 ebss: #
5770042     .int _bss_end # End of not initialized data.
5770043 stack_size: #
5770044     .int 0x8000 # Requested stack size. Every
5770045                 # single application
5770046                 # might change this value.
5770047     #
5770048     # At the next label, the work begins.
5770049     #
5770050 .align 4
5770051 startup_code:
5770052     #
5770053     # Before the call to the main function, it is
5770054     # necessary to extract the value to assign to the
5770055     # global variable 'environ'. It is described as
5770056     # 'char **environ' and should contain the same
5770057     # address pointed by 'envp'. To get this value,
5770058     # the stack is popped and then pushed again.
5770059     # Please recall that the stack was prepared from
5770060     # the process management, at the 'exec()' system
5770061     # call.
5770062     #
5770063     pop %eax # argc
5770064     pop %ebx # argv
5770065     pop %ecx # envp
5770066     mov %ecx, environ # Variable 'environ' comes from
5770067                       # <unistd.h>.
5770068     push %ecx
5770069     push %ebx
5770070     push %eax
5770071     #
```

```
5770072      # Could it be enough? Of course not!
5770073      # To be able to handle the
5770074      # environment, it must be copied inside the table
5770075      # `_environment_table[][]', that is defined inside
5770076      # <stdlib.h>.
5770077      # To copy the environment it is used the function
5770078      # `_environment_setup()', passing the `envp'
5770079      # pointer.
5770080      #
5770081      push %ecx
5770082      call _environment_setup
5770083      add $4, %esp
5770084      #
5770085      # After the environment copy is done, the value for
5770086      # the traditional variable `environ' is updated, to
5770087      # point to the new array of pointer.
5770088      # The updated value comes from variable
5770089      # `_environment', defined inside <stdlib.h>.
5770090      # Then, also the `argv' contained inside
5770091      # the stack is replaced with the new value.
5770092      #
5770093      mov $_environment, %eax
5770094      mov %eax, environ
5770095      #
5770096      pop %eax          # argc
5770097      pop %ebx         # argv[][]
5770098      pop %ecx         # envp[][]
5770099      mov $_environment, %ecx
5770100      push %ecx
5770101      push %ebx
5770102      push %eax
5770103      #
5770104      # Setup standard I/O streams and at-exit table.
5770105      #
5770106      call _stdio_stream_setup
5770107      call _dirent_directory_stream_setup
5770108      call _atexit_setup
```

```
5770109      #
5770110      # Call the main function. The arguments are
5770111      # already pushed inside the stack.
5770112      #
5770113      call main
5770114      #
5770115      # Save the return value at the symbol 'exit_value'.
5770116      #
5770117      mov  %eax, exit_value
5770118      #
5770119      .align 4
5770120      halt:
5770121      #
5770122      pushl $2                # Size of message.
5770123      pushl $exit_value      # Pointer to the message.
5770124      pushl $6                # SYS_EXIT
5770125      call sys
5770126      add  $4, %esp
5770127      add  $4, %esp
5770128      add  $4, %esp
5770129      #
5770130      jmp halt
5770131      #
5770132      #-----
5770133      .align 4
5770134      .section .rodata
5770135      #
5770136      data_magic:
5770137          .quad 0x6F73333264617461    # os32data [1]
5770138      #
5770139      _data_end:
5770140          .int _bss_end
5770141      #
5770142      # [1] This is placed here just to be the same as the
5770143      #       other file 'crt0.sep.s'.
5770144      #       See the other file for an explanation.
5770145      #-----
```



```
5770146 .align 4
5770147 .section .data
5770148 #
5770149 exit_value:
5770150     .int 0x00000000
5770151 #-----
5770152 .align 4
5770153 .section .bss
```

96.1.13 applic/crt0.sep.s



Si veda la sezione [84.5.2](#).

```
5780001 .extern main
5780002 .extern _stdio_stream_setup
5780003 .extern _dirent_directory_stream_setup
5780004 .extern _atexit_setup
5780005 .extern _environment_setup
5780006 .global startup
5780007 .global _data_end
5780008 #-----
5780009 # Please note that, all segment descriptors are already
5780010 # set from the scheduler, and there is also data inside
5780011 # the stack, so that the call to 'main()' function will
5780012 # result as expected.
5780013 #-----
5780014 # The following statement says that the code will start
5780015 # at "startup" label.
5780016 #-----
5780017 .section .text
5780018 #-----
5780019 startup:
5780020     #
5780021     # Jump after initial data.
5780022     #
5780023     jmp startup_code
5780024     #
```

```
5780025 filler:
5780026     #
5780027     # After four bytes, from the start, there is the
5780028     # magic number and other data.
5780029     #
5780030     .space (0x0004 - (filler - startup))
5780031     #
5780032 magic:
5780033     .quad 0x6F7333326170706C    # os32appl
5780034     ;
5780035 doffset:                #
5780036     .int _text_end        # Data offset from start: at
5780037                          # the end of TEXT.
5780038 etext:                  #
5780039     .int _text_end        # End of code
5780040 edata:                  #
5780041     .int _data_end        # End of initialized data.
5780042 ebss:                   #
5780043     .int _bss_end         # End of not initialized data.
5780044 stack_size:             #
5780045     .int 0x8000           # Requested stack size. Every
5780046                          # single application
5780047                          # might change this value.
5780048     #
5780049     # At the next label, the work begins.
5780050     #
5780051 .align 4
5780052 startup_code:
5780053     #
5780054     # Before the call to the main function, it is
5780055     # necessary to extract the value to assign to the
5780056     # global variable 'environ'. It is described as
5780057     # 'char **environ' and should contain the same
5780058     # address pointed by 'envp'. To get this value, the
5780059     # stack is popped and then pushed again.
5780060     # Please recall that the stack was prepared from
5780061     # the process management, at the 'exec()' system
```

```
5780062     # call.
5780063     #
5780064     pop %eax           # argc
5780065     pop %ebx          # argv
5780066     pop %ecx          # envp
5780067     mov %ecx, environ # Variable 'environ' comes from
5780068                       # <unistd.h>.
5780069     push %ecx
5780070     push %ebx
5780071     push %eax
5780072     #
5780073     # Could it be enough? Of course not! To be able to
5780074     # handle the environment, it must be copied inside
5780075     # the table '_environment_table[][]', that is
5780076     # defined inside <stdlib.h>.
5780077     # To copy the environment it is used the function
5780078     # '_environment_setup()', passing the 'envp'
5780079     # pointer.
5780080     #
5780081     push %ecx
5780082     call _environment_setup
5780083     add $4, %esp
5780084     #
5780085     # After the environment copy is done, the value for
5780086     # the traditional variable 'environ' is updated,
5780087     # to point to the new array of pointer.
5780088     # The updated value comes from variable
5780089     # '_environment', defined inside <stdlib.h>.
5780090     # Then, also the 'argv' contained inside
5780091     # the stack is replaced with the new value.
5780092     #
5780093     mov $_environment, %eax
5780094     mov %eax, environ
5780095     #
5780096     pop %eax           # argc
5780097     pop %ebx          # argv[][]
5780098     pop %ecx          # envp[][]
```

```
5780099     mov  $_environment, %ecx
5780100     push %ecx
5780101     push %ebx
5780102     push %eax
5780103     #
5780104     # Setup standard I/O streams and at-exit table.
5780105     #
5780106     call _stdio_stream_setup
5780107     call _dirent_directory_stream_setup
5780108     call _atexit_setup
5780109     #
5780110     # Call the main function. The arguments are already
5780111     # pushed inside the stack.
5780112     #
5780113     call main
5780114     #
5780115     # Save the return value at the symbol 'exit_value'.
5780116     #
5780117     mov  %eax, exit_value
5780118     #
5780119     .align 4
5780120     halt:
5780121     #
5780122     pushl $2                # Size of message.
5780123     pushl $exit_value      # Pointer to the message.
5780124     pushl $6                # SYS_EXIT
5780125     call sys
5780126     add  $4, %esp
5780127     add  $4, %esp
5780128     add  $4, %esp
5780129     #
5780130     jmp halt
5780131     #
5780132     #-----
5780133     .align 4
5780134     .section .rodata
5780135     #
```

```

5780136 data_magic:
5780137     .quad 0x6F73333264617461    # os32data [1]
5780138 #
5780139 _data_end:
5780140     .int _bss_end
5780141 #
5780142 # [1] This signature is just a place holder, at the
5780143 #     beginning of the data segment, which starts at
5780144 #     address 0x00000000. This is to avoid constant
5780145 #     strings to be placed exactly at the beginning
5780146 #     (and it happened so), where the address is
5780147 #     equal to 'NULL'.
5780148 #-----
5780149 .align 4
5780150 .section .data
5780151 #
5780152 exit_value:
5780153     .int 0x00000000
5780154 #-----
5780155 .align 4
5780156 .section .bss

```

96.1.14 applic/date.c

Si veda la sezione [86.10](#).

```

5790001 #include <unistd.h>
5790002 #include <stdlib.h>
5790003 #include <errno.h>
5790004 #include <time.h>
5790005 #include <ctype.h>
5790006 //-----
5790007 static void usage (void);
5790008 //-----
5790009 int
5790010 main (int argc, char *argv[], char *envp[])
5790011 {

```



```
5790012 struct tm *timeptr;
5790013 char string[5];
5790014 time_t timer;
5790015 int length;
5790016 char *input;
5790017 int i;
5790018 int status;
5790019 //
5790020 // There can be at most an argument.
5790021 //
5790022 if (argc > 2)
5790023 {
5790024     usage ();
5790025     return (1);
5790026 }
5790027 //
5790028 // Check if there is no argument: must show the
5790029 // date.
5790030 //
5790031 if (argc == 1)
5790032 {
5790033     timer = time (NULL);
5790034     printf ("%s\n", ctime (&timer));
5790035     return (0);
5790036 }
5790037 //
5790038 // There is one argument and must be the date do
5790039 // set.
5790040 //
5790041 input = argv[1];
5790042 //
5790043 // First get current date, for default values.
5790044 //
5790045 timer = time (NULL);
5790046 timeptr = gmtime (&timer);
5790047 //
5790048 // Verify to have a correct input.
```

```
5790049 //
5790050 length = (int) strlen (input);
5790051 if (length == 8 || length == 10 || length == 12)
5790052 {
5790053     for (i = 0; i < length; i++)
5790054     {
5790055         if (!isdigit (input[i]))
5790056         {
5790057             usage ();
5790058             return (2);
5790059         }
5790060     }
5790061 }
5790062 else
5790063 {
5790064     printf ("input: \"%s\"; length: %i\n", input, length);
5790065     usage ();
5790066     return (3);
5790067 }
5790068 //
5790069 // Select the month.
5790070 //
5790071 string[0] = input[0];
5790072 string[1] = input[1];
5790073 string[2] = '\0';
5790074 timeptr->tm_mon = atoi (string);
5790075 //
5790076 // Select the day.
5790077 //
5790078 string[0] = input[2];
5790079 string[1] = input[3];
5790080 string[2] = '\0';
5790081 timeptr->tm_mday = atoi (string);
5790082 //
5790083 // Select the hour.
5790084 //
5790085 string[0] = input[4];
```

```
5790086 string[1] = input[5];
5790087 string[2] = '\0';
5790088 timeptr->tm_hour = atoi (string);
5790089 //
5790090 // Select the minute.
5790091 //
5790092 string[0] = input[6];
5790093 string[1] = input[7];
5790094 string[2] = '\0';
5790095 timeptr->tm_min = atoi (string);
5790096 //
5790097 // Select the year: must verify if there is a
5790098 // century.
5790099 //
5790100 if (length == 12)
5790101 {
5790102     string[0] = input[8];
5790103     string[1] = input[9];
5790104     string[2] = input[10];
5790105     string[3] = input[11];
5790106     string[4] = '\0';
5790107     timeptr->tm_year = atoi (string);
5790108 }
5790109 else if (length == 10)
5790110 {
5790111     sprintf (string, "%04i", timeptr->tm_year);
5790112     string[2] = input[8];
5790113     string[3] = input[9];
5790114     string[4] = '\0';
5790115     timeptr->tm_year = atoi (string);
5790116 }
5790117 //
5790118 // Now convert to 'time_t'.
5790119 //
5790120 timer = mktime (timeptr);
5790121 //
5790122 // Save to the system.
```



```
5790123 //
5790124 status = stime (&timer);
5790125 if (status != 0)
5790126 {
5790127     perror (NULL);
5790128 }
5790129 //
5790130 return (0);
5790131 }
5790132
5790133 //-----
5790134 static void
5790135 usage (void)
5790136 {
5790137     fprintf (stderr, "Usage: date [MMDDHHMM[[CC]YY]]\n");
5790138 }
```

96.1.15 applic/ed.c

Si veda la sezione 86.11.

```
5800001 //-----
5800002 // 2009.08.18
5800003 // Modified by Daniele Giacomini for 'os16', to
5800004 // harmonize with it, even, when possible, on coding
5800005 // style.
5800006 //
5800007 // The original was taken form ELKS sources:
5800008 // 'elkscmd/misc_utils/ed.c'.
5800009 //-----
5800010 //
5800011 // Copyright (c) 1993 by David I. Bell
5800012 // Permission is granted to use, distribute, or modify
5800013 // this source, provided that this copyright notice
5800014 // remains intact.
5800015 //
5800016 // The "ed" built-in command (much simplified)
```



```
5800017 //
5800018 //-----
5800019
5800020 #include <stdio.h>
5800021 #include <ctype.h>
5800022 #include <unistd.h>
5800023 #include <stdbool.h>
5800024 #include <string.h>
5800025 #include <stdlib.h>
5800026 #include <fcntl.h>
5800027 //-----
5800028 #define isoctal(ch)  (((ch) >= '0') && ((ch) <= '7'))
5800029 #define USERSIZE    1024          /* max line length typed in
5800030                                   by user */
5800031 #define INITBUFSIZE 1024          /* initial buffer size */
5800032 //-----
5800033 typedef int num_t;
5800034 typedef int len_t;
5800035 //
5800036 // The following is the type definition of structure
5800037 // 'line_t', but the structure contains pointers to the
5800038 // same kind of type. With the compiler Bcc, it is the
5800039 // only way to declare it.
5800040 //
5800041 typedef struct line line_t;
5800042 //
5800043 struct line
5800044 {
5800045     line_t *next;
5800046     line_t *prev;
5800047     len_t len;
5800048     char data[1];
5800049 };
5800050 //
5800051 static line_t lines;
5800052 static line_t *curline;
5800053 static num_t curnum;
```

```
5800054 static num_t lastnum;
5800055 static num_t marks[26];
5800056 static bool dirty;
5800057 static char *filename;
5800058 static char searchstring[USERSIZE];
5800059 //
5800060 static char *bufbase;
5800061 static char *bufptr;
5800062 static len_t bufused;
5800063 static len_t bufsize;
5800064 //-----
5800065 static void docommands (void);
5800066 static void subcommand (char *cp, num_t num1, num_t num2);
5800067 static bool getnum (char **retcp, bool * rethavenum,
5800068                   num_t * retnum);
5800069 static bool setcurnum (num_t num);
5800070 static bool initedit (void);
5800071 static void termedit (void);
5800072 static void addlines (num_t num);
5800073 static bool insertline (num_t num, char *data, len_t len);
5800074 static bool deletelines (num_t num1, num_t num2);
5800075 static bool printlines (num_t num1, num_t num2,
5800076                       bool expandflag);
5800077 static bool writelines (char *file, num_t num1, num_t num2);
5800078 static bool readlines (char *file, num_t num);
5800079 static num_t searchlines (char *str, num_t num1,
5800080                          num_t num2);
5800081 static len_t findstring (line_t * lp, char *str,
5800082                        len_t len, len_t offset);
5800083 static line_t *findline (num_t num);
5800084 //-----
5800085 // Main.
5800086 //-----
5800087 int
5800088 main (int argc, char *argv[], char *envp[])
5800089 {
5800090     if (!initedit ())
```

```
5800091     return (2);
5800092     //
5800093     if (argc > 1)
5800094     {
5800095         filename = strdup (argv[1]);
5800096         if (filename == NULL)
5800097         {
5800098             fprintf (stderr, "No memory\n");
5800099             termedit ();
5800100             return (1);
5800101         }
5800102         //
5800103         if (!readlines (filename, 1))
5800104         {
5800105             termedit ();
5800106             return (0);
5800107         }
5800108         //
5800109         if (lastnum)
5800110             setcurnum (1);
5800111         //
5800112         dirty = false;
5800113     }
5800114     //
5800115     docommands ();
5800116     //
5800117     termedit ();
5800118     return (0);
5800119 }
5800120
5800121 //-----
5800122 // Read commands until we are told to stop.
5800123 //-----
5800124 void
5800125 docommands (void)
5800126 {
5800127     char *cp;
```

```
5800128     int len;
5800129     num_t num1;
5800130     num_t num2;
5800131     bool have1;
5800132     bool have2;
5800133     char buf[USERSIZE];
5800134     //
5800135     while (true)
5800136     {
5800137         printf (": ");
5800138         fflush (stdout);
5800139         //
5800140         if (fgets (buf, sizeof (buf), stdin) == NULL)
5800141             {
5800142                 return;
5800143             }
5800144         //
5800145         len = strlen (buf);
5800146         if (len == 0)
5800147             {
5800148                 return;
5800149             }
5800150         //
5800151         cp = &buf[len - 1];
5800152         if (*cp != '\n')
5800153             {
5800154                 fprintf (stderr, "Command line too long\n");
5800155                 do
5800156                     {
5800157                         len = fgetc (stdin);
5800158                     }
5800159                 while ((len != EOF) && (len != '\n'));
5800160                 //
5800161                 continue;
5800162             }
5800163         //
5800164         while ((cp > buf) && isblank (cp[-1]))
```

```
5800165     {
5800166         cp--;
5800167     }
5800168     //
5800169     *cp = '\0';
5800170     //
5800171     cp = buf;
5800172     //
5800173     while (isblank (*cp))
5800174     {
5800175         // *cp++;
5800176         cp++;
5800177     }
5800178     //
5800179     have1 = false;
5800180     have2 = false;
5800181     //
5800182     if ((curnum == 0) && (lastnum > 0))
5800183     {
5800184         curnum = 1;
5800185         curline = lines.next;
5800186     }
5800187     //
5800188     if (!getnum (&cp, &have1, &num1))
5800189     {
5800190         continue;
5800191     }
5800192     //
5800193     while (isblank (*cp))
5800194     {
5800195         cp++;
5800196     }
5800197     //
5800198     if (*cp == ',')
5800199     {
5800200         cp++;
5800201         if (!getnum (&cp, &have2, &num2))
```

```
5800202         {
5800203             continue;
5800204         }
5800205         //
5800206         if (!have1)
5800207         {
5800208             num1 = 1;
5800209         }
5800210         if (!have2)
5800211         {
5800212             num2 = lastnum;
5800213         }
5800214         have1 = true;
5800215         have2 = true;
5800216     }
5800217     //
5800218     if (!have1)
5800219     {
5800220         num1 = curnum;
5800221     }
5800222     if (!have2)
5800223     {
5800224         num2 = num1;
5800225     }
5800226     //
5800227     // Command interpretation switch.
5800228     //
5800229     switch (*cp++)
5800230     {
5800231     case 'a':
5800232         addlines (num1 + 1);
5800233         break;
5800234         //
5800235     case 'c':
5800236         deletelines (num1, num2);
5800237         addlines (num1);
5800238         break;
```

```
5800239 //
5800240 case 'd':
5800241     deletelines (num1, num2);
5800242     break;
5800243 //
5800244 case 'f':
5800245     if (*cp && !isblank (*cp))
5800246     {
5800247         fprintf (stderr, "Bad file command\n");
5800248         break;
5800249     }
5800250 //
5800251 while (isblank (*cp))
5800252     {
5800253         cp++;
5800254     }
5800255 if (*cp == '\0')
5800256     {
5800257         if (filename)
5800258         {
5800259             printf ("\n%s\n\n", filename);
5800260         }
5800261         else
5800262         {
5800263             printf ("No filename\n");
5800264         }
5800265         break;
5800266     }
5800267 //
5800268 cp = strdup (cp);
5800269 //
5800270 if (cp == NULL)
5800271     {
5800272         fprintf (stderr, "No memory for filename\n");
5800273         break;
5800274     }
5800275 //
```



```
5800276         if (filename)
5800277             {
5800278                 free (filename);
5800279             }
5800280             //
5800281             filename = cp;
5800282             break;
5800283             //
5800284         case 'i':
5800285             addlines (num1);
5800286             break;
5800287             //
5800288         case 'k':
5800289             while (isblank (*cp))
5800290                 {
5800291                     cp++;
5800292                 }
5800293             //
5800294             if ((*cp < 'a') || (*cp > 'a') || cp[1])
5800295                 {
5800296                     fprintf (stderr, "Bad mark name\n");
5800297                     break;
5800298                 }
5800299             //
5800300             marks[*cp - 'a'] = num2;
5800301             break;
5800302             //
5800303         case 'l':
5800304             printlines (num1, num2, true);
5800305             break;
5800306             //
5800307         case 'p':
5800308             printlines (num1, num2, false);
5800309             break;
5800310             //
5800311         case 'q':
5800312             while (isblank (*cp))
```

```
5800313         {
5800314             cp++;
5800315         }
5800316         //
5800317         if (have1 || *cp)
5800318         {
5800319             fprintf (stderr, "Bad quit command\n");
5800320             break;
5800321         }
5800322         //
5800323         if (!dirty)
5800324         {
5800325             return;
5800326         }
5800327         //
5800328         printf ("Really quit? ");
5800329         fflush (stdout);
5800330         //
5800331         buf[0] = '\0';
5800332         fgets (buf, sizeof (buf), stdin);
5800333         cp = buf;
5800334         //
5800335         while (isblank (*cp))
5800336         {
5800337             cp++;
5800338         }
5800339         //
5800340         if ((*cp == 'y') || (*cp == 'Y'))
5800341         {
5800342             return;
5800343         }
5800344         //
5800345         break;
5800346         //
5800347         case 'r':
5800348             if (*cp && !isblank (*cp))
5800349             {
```

```
5800350         fprintf (stderr, "Bad read command\n");
5800351         break;
5800352     }
5800353     //
5800354     while (isblank (*cp))
5800355     {
5800356         cp++;
5800357     }
5800358     //
5800359     if (*cp == '\\0')
5800360     {
5800361         fprintf (stderr, "No filename\n");
5800362         break;
5800363     }
5800364     //
5800365     if (!have1)
5800366     {
5800367         num1 = lastnum;
5800368     }
5800369     //
5800370     // Open the file and add to the buffer
5800371     // at the next line.
5800372     //
5800373     if (readlines (cp, num1 + 1))
5800374     {
5800375         //
5800376         // If the file open fails, just
5800377         // break the command.
5800378         //
5800379         break;
5800380     }
5800381     //
5800382     // Set the default file name, if no
5800383     // previous name is available.
5800384     //
5800385     if (filename == NULL)
5800386     {
```

```
5800387         filename = strdup (cp);
5800388     }
5800389     //
5800390     break;
5800391
5800392     case 's':
5800393         subcommand (cp, num1, num2);
5800394         break;
5800395     //
5800396     case 'w':
5800397         if (*cp && !isblank (*cp))
5800398         {
5800399             fprintf (stderr, "Bad write command\n");
5800400             break;
5800401         }
5800402     //
5800403     while (isblank (*cp))
5800404     {
5800405         cp++;
5800406     }
5800407     //
5800408     if (!have1)
5800409     {
5800410         num1 = 1;
5800411         num2 = lastnum;
5800412     }
5800413     //
5800414     // If the file name is not specified, use
5800415     // the
5800416     // default one.
5800417     //
5800418     if (*cp == '\0')
5800419     {
5800420         cp = filename;
5800421     }
5800422     //
5800423     // If even the default file name is not
```

```
5800424     // specified,
5800425     // tell it.
5800426     //
5800427     if (cp == NULL)
5800428     {
5800429         fprintf (stderr, "No file name specified\n");
5800430         break;
5800431     }
5800432     //
5800433     // Write the file.
5800434     //
5800435     writelines (cp, num1, num2);
5800436     //
5800437     break;
5800438     //
5800439 case 'z':
5800440     switch (*cp)
5800441     {
5800442     case '-':
5800443         printlines (curnum - 21, curnum, false);
5800444         break;
5800445     case '.':
5800446         printlines (curnum - 11, curnum + 10, false);
5800447         break;
5800448     default:
5800449         printlines (curnum, curnum + 21, false);
5800450         break;
5800451     }
5800452     break;
5800453     //
5800454 case '.':
5800455     if (have1)
5800456     {
5800457         fprintf (stderr, "No arguments allowed\n");
5800458         break;
5800459     }
5800460     printlines (curnum, curnum, false);
```

```
5800461         break;
5800462         //
5800463     case '-':
5800464         if (setcurnum (curnum - 1))
5800465             {
5800466                 printlines (curnum, curnum, false);
5800467             }
5800468         break;
5800469         //
5800470     case '=':
5800471         printf ("%d\n", num1);
5800472         break;
5800473         //
5800474     case '\0':
5800475         if (have1)
5800476             {
5800477                 printlines (num2, num2, false);
5800478                 break;
5800479             }
5800480         //
5800481         if (setcurnum (curnum + 1))
5800482             {
5800483                 printlines (curnum, curnum, false);
5800484             }
5800485         break;
5800486         //
5800487     default:
5800488         fprintf (stderr, "Unimplemented command\n");
5800489         break;
5800490     }
5800491 }
5800492 }
5800493
5800494 //-----
5800495 // Do the substitute command.
5800496 // The current line is set to the last substitution
5800497 // done.
```

```
5800498 //-----
5800499 void
5800500 subcommand (char *cp, num_t num1, num_t num2)
5800501 {
5800502     int delim;
5800503     char *oldstr;
5800504     char *newstr;
5800505     len_t oldlen;
5800506     len_t newlen;
5800507     len_t deltalen;
5800508     len_t offset;
5800509     line_t *lp;
5800510     line_t *nlp;
5800511     bool globalflag;
5800512     bool printflag;
5800513     bool didsub;
5800514     bool needprint;
5800515
5800516     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
5800517     {
5800518         fprintf (stderr, "Bad line range for substitute\n");
5800519         return;
5800520     }
5800521     //
5800522     globalflag = false;
5800523     printflag = false;
5800524     didsub = false;
5800525     needprint = false;
5800526     //
5800527     if (isblank (*cp) || (*cp == '\\0'))
5800528     {
5800529         fprintf (stderr, "Bad delimiter for substitute\n");
5800530         return;
5800531     }
5800532     //
5800533     delim = *cp++;
5800534     oldstr = cp;
```

```
5800535 //
5800536 cp = strchr (cp, delim);
5800537 //
5800538 if (cp == NULL)
5800539     {
5800540         fprintf (stderr,
5800541                 "Missing 2nd delimiter for " "substitute\n");
5800542         return;
5800543     }
5800544 //
5800545 *cp++ = '\0';
5800546 //
5800547 newstr = cp;
5800548 cp = strchr (cp, delim);
5800549 //
5800550 if (cp)
5800551     {
5800552         *cp++ = '\0';
5800553     }
5800554 else
5800555     {
5800556         cp = "";
5800557     }
5800558 while (*cp)
5800559     {
5800560         switch (*cp++)
5800561         {
5800562             case 'g':
5800563                 globalflag = true;
5800564                 break;
5800565                 //
5800566             case 'p':
5800567                 printflag = true;
5800568                 break;
5800569                 //
5800570             default:
5800571                 fprintf (stderr,
```



```
5800572         "Unknown option for substitute\n");
5800573         return;
5800574     }
5800575 }
5800576 //
5800577 if (*oldstr == '\0')
5800578 {
5800579     if (searchstring[0] == '\0')
5800580     {
5800581         fprintf (stderr, "No previous search string\n");
5800582         return;
5800583     }
5800584     oldstr = searchstring;
5800585 }
5800586 //
5800587 if (oldstr != searchstring)
5800588 {
5800589     strcpy (searchstring, oldstr);
5800590 }
5800591 //
5800592 lp = findline (num1);
5800593 if (lp == NULL)
5800594 {
5800595     return;
5800596 }
5800597 //
5800598 oldlen = strlen (oldstr);
5800599 newlen = strlen (newstr);
5800600 deltalen = newlen - oldlen;
5800601 offset = 0;
5800602 //
5800603 while (num1 <= num2)
5800604 {
5800605     offset = findstring (lp, oldstr, oldlen, offset);
5800606     if (offset < 0)
5800607     {
5800608         if (needprint)
```

```
5800609         {
5800610             printlines (num1, num1, false);
5800611             needprint = false;
5800612         }
5800613         //
5800614         offset = 0;
5800615         lp = lp->next;
5800616         num1++;
5800617         continue;
5800618     }
5800619     //
5800620     needprint = printflag;
5800621     didsub = true;
5800622     dirty = true;
5800623
5800624     // -----
5800625     // If the replacement string is the same size or
5800626     // shorter
5800627     // than the old string, then the substitution is
5800628     // easy.
5800629     // -----
5800630
5800631     if (deltalen <= 0)
5800632     {
5800633         memcpy (&lp->data[offset], newstr, newlen);
5800634         //
5800635         if (deltalen)
5800636         {
5800637             memcpy (&lp->data[offset + newlen],
5800638                 &lp->data[offset + oldlen],
5800639                 lp->len - offset - oldlen);
5800640             //
5800641             lp->len += deltalen;
5800642         }
5800643         //
5800644         offset += newlen;
5800645         //
```

```
5800646         if (globalflag)
5800647             {
5800648                 continue;
5800649             }
5800650         //
5800651         if (needprint)
5800652             {
5800653             printlines (num1, num1, false);
5800654             needprint = false;
5800655             }
5800656         //
5800657         lp = nlp->next;
5800658         num1++;
5800659         continue;
5800660     }
5800661
5800662     // -----
5800663     // The new string is larger, so allocate a new
5800664     // line structure and use that.
5800665     // Link it in place of the old line structure.
5800666     // -----
5800667
5800668     nlp =
5800669         (line_t *) malloc (sizeof (line_t) + lp->len +
5800670                          deltalen);
5800671     //
5800672     if (nlp == NULL)
5800673     {
5800674         fprintf (stderr, "Cannot get memory for line\n");
5800675         return;
5800676     }
5800677     //
5800678     nlp->len = lp->len + deltalen;
5800679     //
5800680     memcpy (nlp->data, lp->data, offset);
5800681     //
5800682     memcpy (&nlp->data[offset], newstr, newlen);
```

```
5800683      //
5800684      memcpy (&nlp->data[offset + newlen],
5800685              &lp->data[offset + oldlen],
5800686              lp->len - offset - oldlen);
5800687      //
5800688      nlp->next = lp->next;
5800689      nlp->prev = lp->prev;
5800690      nlp->prev->next = nlp;
5800691      nlp->next->prev = nlp;
5800692      //
5800693      if (curline == lp)
5800694          {
5800695              curline = nlp;
5800696          }
5800697      //
5800698      free (lp);
5800699      lp = nlp;
5800700      //
5800701      offset += newlen;
5800702      //
5800703      if (globalflag)
5800704          {
5800705              continue;
5800706          }
5800707      //
5800708      if (needprint)
5800709          {
5800710              printlines (num1, num1, false);
5800711              needprint = false;
5800712          }
5800713      //
5800714      lp = lp->next;
5800715      num1++;
5800716      }
5800717      //
5800718      if (!didsub)
5800719          {
```

```
5800720     fprintf (stderr,
5800721             "No substitutions found for \"%s\\\"\\n",
5800722             oldstr);
5800723     }
5800724 }
5800725
5800726 //-----
5800727 // Search a line for the specified string starting at
5800728 // the specified offset in the line. Returns the
5800729 // offset of the found string, or -1.
5800730 //-----
5800731 len_t
5800732 findstring (line_t * lp, char *str, len_t len, len_t offset)
5800733 {
5800734     len_t left;
5800735     char *cp;
5800736     char *ncp;
5800737     //
5800738     cp = &lp->data[offset];
5800739     left = lp->len - offset;
5800740     //
5800741     while (left >= len)
5800742     {
5800743         ncp = memchr (cp, *str, left);
5800744         if (ncp == NULL)
5800745         {
5800746             return (len_t) - 1;
5800747         }
5800748         //
5800749         left -= (ncp - cp);
5800750         if (left < len)
5800751         {
5800752             return (len_t) - 1;
5800753         }
5800754         //
5800755         cp = ncp;
5800756         if (memcmp (cp, str, len) == 0)
```

```
5800757     {
5800758         return (len_t) (cp - lp->data);
5800759     }
5800760     //
5800761     cp++;
5800762     left--;
5800763 }
5800764 //
5800765 return (len_t) - 1;
5800766 }
5800767
5800768 //-----
5800769 // Add lines which are typed in by the user.
5800770 // The lines are inserted just before the specified
5800771 // line number.
5800772 // The lines are terminated by a line containing a
5800773 // single dot (ugly!), or by an end of file.
5800774 //-----
5800775 void
5800776 addlines (num_t num)
5800777 {
5800778     int len;
5800779     char buf[USERSIZE + 1];
5800780     //
5800781     while (fgets (buf, sizeof (buf), stdin))
5800782     {
5800783         if ((buf[0] == '.') && (buf[1] == '\n')
5800784             && (buf[2] == '\0'))
5800785         {
5800786             return;
5800787         }
5800788         //
5800789         len = strlen (buf);
5800790         //
5800791         if (len == 0)
5800792         {
5800793             return;
```

```
5800794     }
5800795     //
5800796     if (buf[len - 1] != '\n')
5800797     {
5800798         fprintf (stderr, "Line too long\n");
5800799         //
5800800         do
5800801         {
5800802             len = fgetc (stdin);
5800803         }
5800804         while ((len != EOF) && (len != '\n'));
5800805         //
5800806         return;
5800807     }
5800808     //
5800809     if (!insertline (num++, buf, len))
5800810     {
5800811         return;
5800812     }
5800813 }
5800814 }
5800815
5800816 //-----
5800817 // Parse a line number argument if it is present. This
5800818 // is a sum or difference of numbers, '.', '$', 'x, or
5800819 // a search string.
5800820 // Returns true if successful (whether or not there was
5800821 // a number).
5800822 // Returns false if there was a parsing error, with a
5800823 // message output.
5800824 // Whether there was a number is returned indirectly,
5800825 // as is the number.
5800826 // The character pointer which stopped the scan is also
5800827 // returned.
5800828 //-----
5800829 static bool
5800830 getnum (char **retcp, bool * rethavenum, num_t * retnum)
```

```
5800831 {
5800832     char *cp;
5800833     char *str;
5800834     bool havenum;
5800835     num_t value;
5800836     num_t num;
5800837     num_t sign;
5800838     //
5800839     cp = *retcp;
5800840     havenum = false;
5800841     value = 0;
5800842     sign = 1;
5800843     //
5800844     while (true)
5800845     {
5800846         while (isblank (*cp))
5800847             {
5800848                 cp++;
5800849             }
5800850         //
5800851         switch (*cp)
5800852         {
5800853             case '.':
5800854                 havenum = true;
5800855                 num = curnum;
5800856                 cp++;
5800857                 break;
5800858             //
5800859             case '$':
5800860                 havenum = true;
5800861                 num = lastnum;
5800862                 cp++;
5800863                 break;
5800864             //
5800865             case '\\':
5800866                 cp++;
5800867                 if ((*cp < 'a') || (*cp > 'z'))
```



```
5800868         {
5800869             fprintf (stderr, "Bad mark name\n");
5800870             return false;
5800871         }
5800872         //
5800873         havenum = true;
5800874         num = marks[*cp++ - 'a'];
5800875         break;
5800876         //
5800877     case '/':
5800878         str = ++cp;
5800879         cp = strchr (str, '/');
5800880         if (cp)
5800881             {
5800882                 *cp++ = '\0';
5800883             }
5800884         else
5800885             {
5800886                 cp = "";
5800887             }
5800888         num = searchlines (str, curnum, lastnum);
5800889         if (num == 0)
5800890             {
5800891                 return false;
5800892             }
5800893         //
5800894         havenum = true;
5800895         break;
5800896         //
5800897     default:
5800898         if (!isdigit (*cp))
5800899             {
5800900                 *retcp = cp;
5800901                 *rethavenum = havenum;
5800902                 *retnum = value;
5800903                 return true;
5800904             }
```

```
5800905         //
5800906         num = 0;
5800907         while (isdigit (*cp))
5800908             {
5800909                 num = num * 10 + *cp++ - '0';
5800910             }
5800911         havenum = true;
5800912         break;
5800913     }
5800914     //
5800915     value += num * sign;
5800916     //
5800917     while (isblank (*cp))
5800918         {
5800919             cp++;
5800920         }
5800921     //
5800922     switch (*cp)
5800923     {
5800924         case '-':
5800925             sign = -1;
5800926             cp++;
5800927             break;
5800928         //
5800929         case '+':
5800930             sign = 1;
5800931             cp++;
5800932             break;
5800933         //
5800934         default:
5800935             *retcp = cp;
5800936             *rethavenum = havenum;
5800937             *retnum = value;
5800938             return true;
5800939     }
5800940 }
5800941 }
```

```
5800942
5800943 //-----
5800944 // Initialize everything for editing.
5800945 //-----
5800946 bool
5800947 initedit (void)
5800948 {
5800949     int i;
5800950     //
5800951     bufsize = INITBUFSIZE;
5800952     bufbase = malloc (bufsize);
5800953     //
5800954     if (bufbase == NULL)
5800955     {
5800956         fprintf (stderr, "No memory for buffer\n");
5800957         return false;
5800958     }
5800959     //
5800960     bufptr = bufbase;
5800961     bufused = 0;
5800962     //
5800963     lines.next = &lines;
5800964     lines.prev = &lines;
5800965     //
5800966     curline = NULL;
5800967     curnum = 0;
5800968     lastnum = 0;
5800969     dirty = false;
5800970     filename = NULL;
5800971     searchstring[0] = '\0';
5800972     //
5800973     for (i = 0; i < 26; i++)
5800974     {
5800975         marks[i] = 0;
5800976     }
5800977     //
5800978     return true;
```

```
5800979 }
5800980
5800981 //-----
5800982 // Finish editing.
5800983 //-----
5800984 void
5800985 termedit (void)
5800986 {
5800987     if (bufbase)
5800988         free (bufbase);
5800989     bufbase = NULL;
5800990     //
5800991     bufptr = NULL;
5800992     bufsize = 0;
5800993     bufused = 0;
5800994     //
5800995     if (filename)
5800996         free (filename);
5800997     filename = NULL;
5800998     //
5800999     searchstring[0] = '\0';
5801000     //
5801001     if (lastnum)
5801002         deletelines (1, lastnum);
5801003     //
5801004     lastnum = 0;
5801005     curnum = 0;
5801006     curline = NULL;
5801007 }
5801008
5801009 //-----
5801010 // Read lines from a file at the specified line number.
5801011 // Returns true if the file was successfully read.
5801012 //-----
5801013 bool
5801014 readlines (char *file, num_t num)
5801015 {
```

```
5801016     int fd;
5801017     int cc;
5801018     len_t len;
5801019     len_t linecount;
5801020     len_t charcount;
5801021     char *cp;
5801022     //
5801023     if ((num < 1) || (num > lastnum + 1))
5801024     {
5801025         fprintf (stderr, "Bad line for read\n");
5801026         return false;
5801027     }
5801028     //
5801029     fd = open (file, O_RDONLY);
5801030     if (fd < 0)
5801031     {
5801032         perror (file);
5801033         return false;
5801034     }
5801035     //
5801036     bufptr = bufbase;
5801037     bufused = 0;
5801038     linecount = 0;
5801039     charcount = 0;
5801040     //
5801041     printf ("\n%s\n", "", file);
5801042     fflush (stdout);
5801043     //
5801044     do
5801045     {
5801046         cp = memchr (bufptr, '\n', bufused);
5801047         if (cp)
5801048         {
5801049             len = (cp - bufptr) + 1;
5801050             //
5801051             if (!insertline (num, bufptr, len))
5801052             {
```

```
5801053         close (fd);
5801054         return false;
5801055     }
5801056     //
5801057     bufptr += len;
5801058     bufused -= len;
5801059     charcount += len;
5801060     linecount++;
5801061     num++;
5801062     continue;
5801063 }
5801064 //
5801065 if (bufptr != bufbase)
5801066 {
5801067     memcpy (bufbase, bufptr, bufused);
5801068     bufptr = bufbase + bufused;
5801069 }
5801070 //
5801071 if (bufused >= bufsize)
5801072 {
5801073     len = (bufsize * 3) / 2;
5801074     cp = realloc (bufbase, len);
5801075     if (cp == NULL)
5801076     {
5801077         fprintf (stderr, "No memory for buffer\n");
5801078         close (fd);
5801079         return false;
5801080     }
5801081     //
5801082     bufbase = cp;
5801083     bufptr = bufbase + bufused;
5801084     bufsize = len;
5801085 }
5801086 //
5801087 cc = read (fd, bufptr, bufsize - bufused);
5801088 bufused += cc;
5801089 bufptr = bufbase;
```

```
5801090     }
5801091     while (cc > 0);
5801092     //
5801093     if (cc < 0)
5801094     {
5801095         perror (file);
5801096         close (fd);
5801097         return false;
5801098     }
5801099     //
5801100     if (bufused)
5801101     {
5801102         if (!insertline (num, bufptr, bufused))
5801103         {
5801104             close (fd);
5801105             return -1;
5801106         }
5801107         linecount++;
5801108         charcount += bufused;
5801109     }
5801110     //
5801111     close (fd);
5801112     //
5801113     printf ("%d lines%s, %d chars\n",
5801114            linecount, (bufused ? " (incomplete)" : ""),
5801115            charcount);
5801116     //
5801117     return true;
5801118 }
5801119
5801120 //-----
5801121 // Write the specified lines out to the specified file.
5801122 // Returns true if successful, or false on an error
5801123 // with a message output.
5801124 //-----
5801125 bool
5801126 writelines (char *file, num_t num1, num_t num2)
```

```
5801127 {
5801128     int fd;
5801129     line_t *lp;
5801130     len_t linecount;
5801131     len_t charcount;
5801132     //
5801133     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
5801134         {
5801135             fprintf (stderr, "Bad line range for write\n");
5801136             return false;
5801137         }
5801138     //
5801139     linecount = 0;
5801140     charcount = 0;
5801141     //
5801142     fd = creat (file, 0666);
5801143     if (fd < 0)
5801144         {
5801145             perror (file);
5801146             return false;
5801147         }
5801148     //
5801149     printf ("\n%s\n", "", file);
5801150     fflush (stdout);
5801151     //
5801152     lp = findline (num1);
5801153     if (lp == NULL)
5801154         {
5801155             close (fd);
5801156             return false;
5801157         }
5801158     //
5801159     while (num1++ <= num2)
5801160         {
5801161             if (write (fd, lp->data, lp->len) != lp->len)
5801162                 {
5801163                     perror (file);
```



```
5801164         close (fd);
5801165         return false;
5801166     }
5801167     //
5801168     charcount += lp->len;
5801169     linecount++;
5801170     lp = lp->next;
5801171 }
5801172 //
5801173 if (close (fd) < 0)
5801174 {
5801175     perror (file);
5801176     return false;
5801177 }
5801178 //
5801179 printf ("%d lines, %d chars\n", linecount, charcount);
5801180 //
5801181 return true;
5801182 }
5801183
5801184 //-----
5801185 // Print lines in a specified range.
5801186 // The last line printed becomes the current line.
5801187 // If expandflag is true, then the line is printed
5801188 // specially to show magic characters.
5801189 //-----
5801190 bool
5801191 printlines (num_t num1, num_t num2, bool expandflag)
5801192 {
5801193     line_t *lp;
5801194     unsigned char *cp;
5801195     int ch;
5801196     len_t count;
5801197     //
5801198     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
5801199     {
5801200         fprintf (stderr, "Bad line range for print\n");
```

```
5801201     return false;
5801202     }
5801203     //
5801204     lp = findline (num1);
5801205     if (lp == NULL)
5801206     {
5801207         return false;
5801208     }
5801209     //
5801210     while (num1 <= num2)
5801211     {
5801212         if (!expandflag)
5801213         {
5801214             write (STDOUT_FILENO, lp->data, lp->len);
5801215             setcurnum (num1++);
5801216             lp = lp->next;
5801217             continue;
5801218         }
5801219
5801220         // -----
5801221         // Show control characters and characters with
5801222         // the high bit set specially.
5801223         // -----
5801224
5801225         cp = (unsigned char *) lp->data;
5801226         count = lp->len;
5801227         //
5801228         if ((count > 0) && (cp[count - 1] == '\n'))
5801229         {
5801230             count--;
5801231         }
5801232         //
5801233         while (count-- > 0)
5801234         {
5801235             ch = *cp++;
5801236             if (ch & 0x80)
5801237             {
```

```
5801238         fputs ("M-", stdout);
5801239         ch &= 0x7f;
5801240     }
5801241     if (ch < ' ')
5801242     {
5801243         fputc ('^', stdout);
5801244         ch += '@';
5801245     }
5801246     if (ch == 0x7f)
5801247     {
5801248         fputc ('^', stdout);
5801249         ch = '?';
5801250     }
5801251     fputc (ch, stdout);
5801252 }
5801253 //
5801254 fputs ("$\n", stdout);
5801255 //
5801256 setcurnum (num1++);
5801257 lp = lp->next;
5801258 }
5801259 //
5801260 return true;
5801261 }
5801262
5801263 //-----
5801264 // Insert a new line with the specified text.
5801265 // The line is inserted so as to become the specified
5801266 // line, thus pushing any existing and further lines
5801267 // down one.
5801268 // The inserted line is also set to become the current
5801269 // line.
5801270 // Returns true if successful.
5801271 //-----
5801272 bool
5801273 insertline (num_t num, char *data, len_t len)
5801274 {
```

```
5801275     line_t *newlp;
5801276     line_t *lp;
5801277     //
5801278     if ((num < 1) || (num > lastnum + 1))
5801279     {
5801280         fprintf (stderr, "Inserting at bad line number\n");
5801281         return false;
5801282     }
5801283     //
5801284     newlp = (line_t *) malloc (sizeof (line_t) + len - 1);
5801285     if (newlp == NULL)
5801286     {
5801287         fprintf (stderr,
5801288             "Failed to allocate memory for line\n");
5801289         return false;
5801290     }
5801291     //
5801292     memcpy (newlp->data, data, len);
5801293     newlp->len = len;
5801294     //
5801295     if (num > lastnum)
5801296     {
5801297         lp = &lines;
5801298     }
5801299     else
5801300     {
5801301         lp = findline (num);
5801302         if (lp == NULL)
5801303         {
5801304             free ((char *) newlp);
5801305             return false;
5801306         }
5801307     }
5801308     //
5801309     newlp->next = lp;
5801310     newlp->prev = lp->prev;
5801311     lp->prev->next = newlp;
```

```
5801312     lp->prev = newlp;
5801313     //
5801314     lastnum++;
5801315     dirty = true;
5801316     //
5801317     return setcurnum (num);
5801318 }
5801319
5801320 //-----
5801321 // Delete lines from the given range.
5801322 //-----
5801323 bool
5801324 deletelines (num_t num1, num_t num2)
5801325 {
5801326     line_t *lp;
5801327     line_t *nlp;
5801328     line_t *plp;
5801329     num_t count;
5801330     //
5801331     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
5801332     {
5801333         fprintf (stderr, "Bad line numbers for delete\n");
5801334         return false;
5801335     }
5801336     //
5801337     lp = findline (num1);
5801338     if (lp == NULL)
5801339     {
5801340         return false;
5801341     }
5801342     //
5801343     if ((curnum >= num1) && (curnum <= num2))
5801344     {
5801345         if (num2 < lastnum)
5801346         {
5801347             setcurnum (num2 + 1);
5801348         }
```

```
5801349     else if (num1 > 1)
5801350     {
5801351         setcurnum (num1 - 1);
5801352     }
5801353     else
5801354     {
5801355         curnum = 0;
5801356     }
5801357 }
5801358 //
5801359 count = num2 - num1 + 1;
5801360 //
5801361 if (curnum > num2)
5801362     {
5801363         curnum -= count;
5801364     }
5801365 //
5801366 lastnum -= count;
5801367 //
5801368 while (count-- > 0)
5801369     {
5801370         nlp = lp->next;
5801371         plp = lp->prev;
5801372         plp->next = nlp;
5801373         nlp->prev = plp;
5801374         lp->next = NULL;
5801375         lp->prev = NULL;
5801376         lp->len = 0;
5801377         free (lp);
5801378         lp = nlp;
5801379     }
5801380 //
5801381 dirty = true;
5801382 //
5801383 return true;
5801384 }
5801385
```

```
5801386 //-----  
5801387 // Search for a line which contains the specified  
5801388 // string.  
5801389 // If the string is NULL, then the previously searched  
5801390 // for string is used. The currently searched for  
5801391 // string is saved for future use.  
5801392 // Returns the line number which matches, or 0 if there  
5801393 // was no match with an error printed.  
5801394 //-----  
5801395 num_t  
5801396 searchlines (char *str, num_t num1, num_t num2)  
5801397 {  
5801398     line_t *lp;  
5801399     int len;  
5801400     //  
5801401     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))  
5801402     {  
5801403         fprintf (stderr, "Bad line numbers for search\n");  
5801404         return 0;  
5801405     }  
5801406     //  
5801407     if (*str == '\\0')  
5801408     {  
5801409         if (searchstring[0] == '\\0')  
5801410         {  
5801411             fprintf (stderr, "No previous search string\n");  
5801412             return 0;  
5801413         }  
5801414         str = searchstring;  
5801415     }  
5801416     //  
5801417     if (str != searchstring)  
5801418     {  
5801419         strcpy (searchstring, str);  
5801420     }  
5801421     //  
5801422     len = strlen (str);
```

```
5801423 //
5801424 lp = findline (num1);
5801425 if (lp == NULL)
5801426     {
5801427         return 0;
5801428     }
5801429 //
5801430 while (num1 <= num2)
5801431     {
5801432         if (findstring (lp, str, len, 0) >= 0)
5801433             {
5801434                 return num1;
5801435             }
5801436         //
5801437         num1++;
5801438         lp = lp->next;
5801439     }
5801440 //
5801441 fprintf (stderr, "Cannot find string \"%s\"\n", str);
5801442 //
5801443 return 0;
5801444 }
5801445
5801446 //-----
5801447 // Return a pointer to the specified line number.
5801448 //-----
5801449 line_t *
5801450 findline (num_t num)
5801451 {
5801452     line_t *lp;
5801453     num_t lnum;
5801454     //
5801455     if ((num < 1) || (num > lastnum))
5801456         {
5801457             fprintf (stderr,
5801458                 "Line number %d does not exist\n", num);
5801459             return NULL;
```



```
5801460     }
5801461     //
5801462     if (curnum <= 0)
5801463     {
5801464         curnum = 1;
5801465         curline = lines.next;
5801466     }
5801467     //
5801468     if (num == curnum)
5801469     {
5801470         return curline;
5801471     }
5801472     //
5801473     lp = curline;
5801474     lnum = curnum;
5801475     //
5801476     if (num < (curnum / 2))
5801477     {
5801478         lp = lines.next;
5801479         lnum = 1;
5801480     }
5801481     else if (num > ((curnum + lastnum) / 2))
5801482     {
5801483         lp = lines.prev;
5801484         lnum = lastnum;
5801485     }
5801486     //
5801487     while (lnum < num)
5801488     {
5801489         lp = lp->next;
5801490         lnum++;
5801491     }
5801492     //
5801493     while (lnum > num)
5801494     {
5801495         lp = lp->prev;
5801496         lnum--;
```

```
5801497     }
5801498     //
5801499     return lp;
5801500 }
5801501
5801502 //-----
5801503 // Set the current line number.
5801504 // Returns true if successful.
5801505 //-----
5801506 bool
5801507 setcurnum (num_t num)
5801508 {
5801509     line_t *lp;
5801510     //
5801511     lp = findline (num);
5801512     if (lp == NULL)
5801513     {
5801514         return false;
5801515     }
5801516     //
5801517     curnum = num;
5801518     curline = lp;
5801519     //
5801520     return true;
5801521 }
5801522
5801523 /* END CODE */
```

96.1.16 applic/getty.c



Si veda la sezione [92.2](#).

```
5810001 #include <unistd.h>
5810002 #include <stdio.h>
5810003 #include <stdlib.h>
5810004 #include <signal.h>
5810005 #include <sys/wait.h>
```

```
5810006 #include <limits.h>
5810007 #include <sys/os32.h>
5810008 #include <fcntl.h>
5810009 #include <stdio.h>
5810010 //-----
5810011 int
5810012 main (int argc, char *argv[], char *envp[])
5810013 {
5810014     char *device_name;
5810015     int fdn;
5810016     char *exec_argv[2];
5810017     char **exec_envp;
5810018     char buffer[BUFSIZ];
5810019     ssize_t size_read;
5810020     int status;
5810021     //
5810022     // The first argument is mandatory and must be a
5810023     // console terminal.
5810024     //
5810025     device_name = argv[1];
5810026     //
5810027     // A console terminal is correctly selected (but it
5810028     // is not checked
5810029     // if it is a really available one).
5810030     // Set as a process group leader.
5810031     //
5810032     setpgrp ();
5810033     //
5810034     // Open the terminal, that should become the
5810035     // controlling terminal:
5810036     // close the standard input and open the new
5810037     // terminal (r/w).
5810038     //
5810039     close (0);
5810040     fdn = open (device_name, O_RDWR);
5810041     if (fdn < 0)
5810042     {
```

```
5810043      //
5810044      // Cannot open terminal. A message should
5810045      // appear, at least
5810046      // to the current console.
5810047      //
5810048      perror (NULL);
5810049      return (-1);
5810050    }
5810051    //
5810052    // Reset terminal device permissions and ownership.
5810053    //
5810054    status = fchown (fdn, (uid_t) 0, (gid_t) 0);
5810055    if (status != 0)
5810056      {
5810057        perror (NULL);
5810058      }
5810059    status = fchmod (fdn, 0644);
5810060    if (status != 0)
5810061      {
5810062        perror (NULL);
5810063      }
5810064    //
5810065    // The terminal is open and it should be already the
5810066    // controlling
5810067    // one: show '/etc/issue'. The same variable 'fdn'
5810068    // is used, because
5810069    // the controlling terminal will never be closed
5810070    // (the exit syscall
5810071    // will do it).
5810072    //
5810073    fdn = open ("/etc/issue", O_RDONLY);
5810074    if (fdn > 0)
5810075      {
5810076        //
5810077        // The file is present and is shown.
5810078        //
5810079        for (size_read = 1; size_read > 0;)
```

```
5810080     {
5810081         size_read =
5810082             read (fdn, buffer, (size_t) (BUFSIZ - 1));
5810083         if (size_read < 0)
5810084             {
5810085                 break;
5810086             }
5810087         buffer[size_read] = '\0';
5810088         printf ("%s", buffer);
5810089     }
5810090     close (fdn);
5810091 }
5810092 //
5810093 // Show the terminal.
5810094 //
5810095 printf ("This is terminal %s\n", device_name);
5810096 //
5810097 // It is time to exec login: the environment is
5810098 // inherited directly
5810099 // from 'init'.
5810100 //
5810101 exec_argv[0] = "login";
5810102 exec_argv[1] = NULL;
5810103 exec_envp = envp;
5810104 execve ("/bin/login", exec_argv, exec_envp);
5810105 //
5810106 // If 'execve()' returns, it is an error.
5810107 //
5810108 exit (-1);
5810109 }
```

96.1.17 applic/http.c

Si veda la sezione [92.3](#).

```
5820001 #include <sys/stat.h>
5820002 #include <sys/types.h>
```

```
5820003 #include <unistd.h>
5820004 #include <stdlib.h>
5820005 #include <fcntl.h>
5820006 #include <errno.h>
5820007 #include <signal.h>
5820008 #include <stdio.h>
5820009 #include <string.h>
5820010 #include <limits.h>
5820011 #include <libgen.h>
5820012 #include <arpa/inet.h>
5820013 #include <sys/socket.h>
5820014 #include <stdint.h>
5820015 #include <stdbool.h>
5820016 //-----
5820017 #define DEBUG 0
5820018 static void usage (void);
5820019 static int send_file (int sfdn2, const char *path);
5820020 static int send_line (int sfdn2, const char *line);
5820021 char buffer[BUFSIZ];
5820022 char path_absolute[PATH_MAX];
5820023 //-----
5820024 int
5820025 main (int argc, char *argv[], char *envp[])
5820026 {
5820027     int opt;
5820028     //extern char *optarg;           // not used.
5820029     extern int optind;
5820030     extern int optopt;
5820031     //
5820032     int status;
5820033     int sfdn;
5820034     int sfdn2;
5820035     struct sockaddr_in sa_local;
5820036     struct sockaddr_in sa_remote;
5820037     socklen_t sa_remote_size = sizeof (struct sockaddr_in);
5820038     ssize_t recv_size;
5820039     char *addr = "0.0.0.0";
```

```
5820040 char *www = NULL;
5820041 char *path = NULL;
5820042 int port;
5820043 bool request_read;
5820044 int b;          // index inside the buffer string
5820045 // buffer
5820046 char *string = NULL;
5820047 struct stat file_status;
5820048 //
5820049 // Check for options: no options at the moment.
5820050 //
5820051 while ((opt = getopt (argc, argv, ":")) != -1)
5820052 {
5820053     switch (opt)
5820054     {
5820055         case '?':
5820056             fprintf (stderr, "Unknown option -%c.\n", optopt);
5820057             usage ();
5820058             return (1);
5820059             break;
5820060         case ':':
5820061             fprintf (stderr,
5820062                     "Missing argument for option -%c\n",
5820063                     optopt);
5820064             usage ();
5820065             return (1);
5820066             break;
5820067         default:
5820068             fprintf (stderr,
5820069                     "Getopt problem: "
5820070                     "unknown option %c\n", opt);
5820071             usage ();
5820072             return (1);
5820073     }
5820074 }
5820075 //
5820076 // Arguments.
```

```
5820077 //
5820078 if (optind == (argc - 2))
5820079 {
5820080 //
5820081 // There are exactly two arguments: the port and
5820082 // the www root path.
5820083 //
5820084 port = atoi (argv[argc - 2]);
5820085 www = argv[argc - 1];
5820086 }
5820087 else
5820088 {
5820089 //
5820090 // Arguments wrong!
5820091 //
5820092 printf ("optind = %i = %s, argc = %i\n", optind,
5820093         argv[optind], argc);
5820094 usage ();
5820095 return (2);
5820096 }
5820097 //
5820098 // Set the local address.
5820099 //
5820100 sa_local.sin_family = AF_INET;
5820101 sa_local.sin_port = htons (port);
5820102 inet_pton (AF_INET, addr, &sa_local.sin_addr.s_addr);
5820103 //
5820104 // Open the socket.
5820105 //
5820106 sfdn = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
5820107 if (sfdn < 0)
5820108 {
5820109     perror (NULL);
5820110     return (3);
5820111 }
5820112 if (DEBUG)
5820113 {
```



```
5820114     printf ("HTTP: listening socket number "
5820115             "is %i.\n", sfdn);
5820116     }
5820117     //
5820118     // Set it listening: bind the local 'sa' location.
5820119     //
5820120     status = bind (sfdn, (struct sockaddr *) &sa_local,
5820121                 sizeof (sa_local));
5820122     if (status < 0)
5820123     {
5820124         perror (NULL);
5820125         close (sfdn);
5820126         return (4);
5820127     }
5820128     //
5820129     // Listen (TCP).
5820130     //
5820131     status = listen (sfdn, 1);
5820132     if (status < 0)
5820133     {
5820134         perror (NULL);
5820135         close (sfdn);
5820136         return (5);
5820137     }
5820138     //
5820139     // Accept connections, inside a loop.
5820140     //
5820141     while (1)
5820142     {
5820143         //
5820144         // Accept.
5820145         //
5820146         if (DEBUG)
5820147         {
5820148             printf
5820149                 ("HTTP: listening socket number is %i.\n",
5820150                 sfdn);
```

```
5820151     }
5820152     //
5820153     sfdn2 =
5820154         accept (sfdn, (struct sockaddr *) &sa_remote,
5820155             &sa_remote_size);
5820156     //
5820157     if (sfdn2 < 0)
5820158     {
5820159         perror (NULL);
5820160         close (sfdn2);
5820161         continue;
5820162     }
5820163     if (DEBUG)
5820164     {
5820165         printf
5820166             ("HTTP: new connection with socket "
5820167             "number %i.\n", sfdn2);
5820168     }
5820169     //
5820170     // Define the socket non blocking.
5820171     //
5820172     status = fcntl (sfdn2, F_SETFL, O_NONBLOCK);
5820173     if (status < 0)
5820174     {
5820175         perror (NULL);
5820176         return (9);
5820177     }
5820178     //
5820179     // Will read from the remote.
5820180     //
5820181     path_absolute[0] = 0;
5820182     request_read = 1;
5820183     while (request_read)
5820184     {
5820185         //
5820186         // Read a line from the remote side.
5820187         //
```

```
5820188     for (b = 0; b < (BUFSIZ - 2); b++, buffer[b] = 0)
5820189     {
5820190         //
5820191         // Read a single character from the
5820192         // remote side.
5820193         //
5820194         recv_size =
5820195             recv (sfdn2, &buffer[b], (size_t) 1, 0);
5820196         //
5820197         if (recv_size < 0)
5820198             {
5820199                 if (errno == EAGAIN
5820200                     || errno == EWOULDBLOCK)
5820201                     {
5820202                         b--;
5820203                         continue;
5820204                     }
5820205                 else
5820206                     {
5820207                         perror (NULL);
5820208                         close (sfdn2);
5820209                         continue;
5820210                     }
5820211             }
5820212         if (recv_size == 0)
5820213             {
5820214                 //
5820215                 // It is the end of stream, but
5820216                 // should not happen.
5820217                 //
5820218                 buffer[b] = 0;
5820219                 request_read = 0;
5820220                 if (DEBUG)
5820221                     {
5820222                         printf ("HTTP: end of stream, "
5820223                             "but should not "
5820224                             "happen here! "
```

```
5820225         "\">%s"\n", buffer);
5820226     }
5820227     break;
5820228 }
5820229 //
5820230 if (buffer[b] == '\r')
5820231 {
5820232     //
5820233     // Ignore CR.
5820234     //
5820235     b--;
5820236     continue;
5820237 }
5820238 //
5820239 if (buffer[b] == '\n')
5820240 {
5820241     //
5820242     // End of line.
5820243     //
5820244     buffer[b] = 0;
5820245     if (DEBUG)
5820246     {
5820247         printf ("HTTP: %s\n", buffer);
5820248     }
5820249     break;
5820250 }
5820251 }
5820252 //
5820253 // Was it the end of the header?
5820254 //
5820255 if (strlen (buffer) == 0)
5820256 {
5820257     //
5820258     // End of header.
5820259     //
5820260     request_read = 0;
5820261     break;
```

```
5820262     }
5820263     //
5820264     // We are reading the header: was it the GET
5820265     // command?
5820266     //
5820267     string = strtok (buffer, " ");
5820268     if (strncmp (string, "GET", 4) == 0)
5820269     {
5820270         //
5820271         // It is a GET: find the path.
5820272         //
5820273         path = strtok (NULL, " ");
5820274         strncat (path_absolute, www, PATH_MAX - 1);
5820275         strncat (path_absolute, path,
5820276                 (PATH_MAX -
5820277                  strlen (path_absolute) - 1));
5820278     }
5820279 }
5820280 //
5820281 // Verify to have received a 'GET' request.
5820282 //
5820283 if (strlen (path_absolute) == 0)
5820284 {
5820285     //
5820286     // There is no path inside the GET command;
5820287     // maybe there is
5820288     // no GET command either: 400
5820289     //
5820290     if (DEBUG)
5820291     {
5820292         printf ("HTTP: 400 Bad Request: "
5820293                "no path inside the GET "
5820294                "command.\n");
5820295     }
5820296     send_line (sfdn2, "HTTP/1.0 400 Bad Request\r\n");
5820297     send_line (sfdn2, "Content-Type: text/html\r\n");
5820298     send_line (sfdn2, "Content-Length: 26\r\n");
```

```
5820299         send_line (sfdn2, "\r\n");
5820300         send_line (sfdn2, "<H1>400 Bad Request</H1>\r\n");
5820301     }
5820302     //
5820303     // Verify the path.
5820304     //
5820305     if (stat (path_absolute, &file_status) != 0)
5820306     {
5820307         //
5820308         // The path inside the GET command does not
5820309         // exists: 404
5820310         //
5820311         if (DEBUG)
5820312         {
5820313             printf ("HTTP: 404 Not Found: "
5820314                    "the path \"%s\" does not "
5820315                    "exists.\n", path_absolute);
5820316         }
5820317         send_line (sfdn2, "HTTP/1.0 404 Not Found\r\n");
5820318         send_line (sfdn2, "Content-Type: text/html\r\n");
5820319         send_line (sfdn2, "Content-Length: 24\r\n");
5820320         send_line (sfdn2, "\r\n");
5820321         send_line (sfdn2, "<H1>404 Not Found</H1>\r\n");
5820322     }
5820323     else
5820324     {
5820325         //
5820326         // File exists: check the file type.
5820327         //
5820328         if (S_ISDIR (file_status.st_mode))
5820329         {
5820330             //
5820331             // Test to find 'index.html'.
5820332             //
5820333             strncat (path_absolute, "index.html",
5820334                     (PATH_MAX -
5820335                      strlen (path_absolute) - 1));
```

```
5820336         //
5820337         if (stat (path_absolute, &file_status) != 0)
5820338             {
5820339                 //
5820340                 // The index file inside the path
5820341                 // requested
5820342                 // does not exists: 404
5820343                 //
5820344                 if (DEBUG)
5820345                     {
5820346                     printf ("HTTP: 404 Not Found: "
5820347                             "the path \"%s\" does "
5820348                             "not exists.\n",
5820349                             path_absolute);
5820350                     }
5820351                 send_line (sfdn2,
5820352                           "HTTP/1.0 404 Not "
5820353                           "Found\r\n");
5820354                 send_line (sfdn2,
5820355                           "Content-Type: "
5820356                           "text/html\r\n");
5820357                 send_line (sfdn2,
5820358                           "Content-Length: 24\r\n");
5820359                 send_line (sfdn2, "\r\n");
5820360                 send_line (sfdn2,
5820361                           "<H1>404 Not Found"
5820362                           "</H1>\r\n");
5820363             }
5820364         }
5820365         //
5820366         // There is a file to send.
5820367         //
5820368         send_file (sfdn2, path_absolute);
5820369     }
5820370     //
5820371     // The socket 'sfdn2' might be already closed;
5820372     // if so, the variable was reset to zero.
```

```
5820373     //
5820374     if (sfdn2 != 0)
5820375         close (sfdn2);
5820376     buffer[0] = 0;
5820377     if (DEBUG)
5820378     {
5820379         printf
5820380             ("HTTP: connection closed: continue "
5820381              "listening.\n");
5820382     }
5820383     continue;
5820384 }
5820385 //
5820386 // All done.
5820387 //
5820388 close (sfdn);
5820389 return (0);
5820390 }
5820391
5820392 //-----
5820393 static int
5820394 send_file (int sfdn2, const char *path)
5820395 {
5820396     // size_t sent_size;
5820397     size_t file_size;
5820398     struct stat file_status;
5820399     char ascii_size[32];
5820400     int fdn;
5820401     char *buffer_in = buffer;
5820402     char *buffer_out;
5820403     ssize_t count_in;     // Read counter.
5820404     ssize_t count_out;   // Write counter.
5820405     //
5820406     if (sfdn2 == 0)
5820407         return (-1);
5820408     //
5820409     if (stat (path, &file_status) != 0)
```



```
5820410     {
5820411         perror (NULL);
5820412         close (sfdn2);
5820413         sfdn2 = 0;
5820414         return (-1);
5820415     }
5820416     //
5820417     file_size = file_status.st_size;
5820418     sprintf (ascii_size, "%i", file_size);
5820419     //
5820420     fdn = open (path, O_RDONLY);
5820421     if (fdn < 0)
5820422     {
5820423         if (DEBUG)
5820424         {
5820425             printf ("HTTP: 403 Forbidden: %s ", path);
5820426         }
5820427         perror (path);
5820428         send_line (sfdn2, "HTTP/1.0 403 Forbidden\r\n");
5820429         send_line (sfdn2, "Content-Type: text/html\r\n");
5820430         send_line (sfdn2, "Content-Length: 24\r\n");
5820431         send_line (sfdn2, "\r\n");
5820432         send_line (sfdn2, "<H1>404 Forbidden</H1>\r\n");
5820433         close (sfdn2);
5820434         sfdn2 = 0;
5820435         return (-1);
5820436     }
5820437     //
5820438     send_line (sfdn2, "HTTP/1.0 200 OK\r\n");
5820439     send_line (sfdn2, "Content-Type: text/html\r\n");
5820440     send_line (sfdn2, "Content-Length: ");
5820441     send_line (sfdn2, ascii_size);
5820442     send_line (sfdn2, "\r\n");
5820443     send_line (sfdn2, "\r\n");
5820444     //
5820445     // Copy the data.
5820446     //
```

```
5820447 while (1)
5820448     {
5820449         count_in = read (fdn, buffer_in, (size_t) BUFSIZ);
5820450         if (count_in > 0)
5820451             {
5820452                 for (buffer_out = buffer_in; count_in > 0;)
5820453                     {
5820454                         count_out = send (sfdn2, buffer_out,
5820455                                           (size_t) count_in, 0);
5820456                         if (count_out < 0)
5820457                             {
5820458                                 if (errno == EAGAIN
5820459                                     || errno == EWOULDBLOCK)
5820460                                     {
5820461                                         continue;
5820462                                     }
5820463                                 else
5820464                                     {
5820465                                     fprintf (stderr,
5820466                                             "[HTTP] cannot "
5820467                                             "send 1!\n");
5820468                                     perror (NULL);
5820469                                     close (fdn);
5820470                                     close (sfdn2);
5820471                                     sfdn2 = 0;
5820472                                     return (-1);
5820473                                     }
5820474                             }
5820475                     }
5820476                     //
5820477                     // If not all data is written, continue
5820478                     // writing, but change the buffer start
5820479                     // position and the
5820480                     // amount to be written.
5820481                     //
5820481                     buffer_out += count_out;
5820482                     count_in -= count_out;
5820483                 }
            }
```

```
5820484     }
5820485     else if (count_in < 0)
5820486     {
5820487         perror (path);
5820488         close (fdn);
5820489         close (sfdn2);
5820490         sfdn2 = 0;
5820491         return (-1);
5820492     }
5820493     else
5820494     {
5820495         break;
5820496     }
5820497 }
5820498 //
5820499 return (0);
5820500 }
5820501
5820502 //-----
5820503 static int
5820504 send_line (int sfdn2, const char *line)
5820505 {
5820506     size_t sent_size;
5820507     size_t line_size = strlen (line);
5820508     const char *start = line;
5820509     //
5820510     if (sfdn2 == 0)
5820511         return (-1);
5820512     //
5820513     while (1)
5820514     {
5820515         errno = 0;
5820516         sent_size =
5820517             send (sfdn2, start, (size_t) line_size, 0);
5820518         //
5820519         //
5820520         //
```

```
5820521     if (DEBUG)
5820522     {
5820523         printf
5820524             ("[HTTP] line_size=%i, sent_size=%i, "
5820525              "error=%i\n",
5820526              (int) line_size, (int) sent_size, errno);
5820527     }
5820528     if (sent_size < 0)
5820529     {
5820530         if (errno == EAGAIN || errno == EWOULDBLOCK)
5820531         {
5820532             continue;
5820533         }
5820534         else
5820535         {
5820536             fprintf (stderr, "[HTTP] cannot send 2!\n");
5820537             perror (NULL);
5820538             close (sfdn2);
5820539             sfdn2 = 0;
5820540             return (-1);
5820541         }
5820542     }
5820543     //
5820544     //
5820545     //
5820546     if (sent_size < line_size)
5820547     {
5820548         start = &start[sent_size];
5820549         line_size -= sent_size;
5820550         //
5820551         continue;
5820552     }
5820553     return (0);
5820554 }
5820555 }
5820556
5820557 //-----
```

```
5820558 static void
5820559 usage (void)
5820560 {
5820561     fprintf (stderr,
5820562             "os32 http usage:\n"
5820563             "\n"
5820564             "http PORT WWW_ROOT_PATH\n"
5820565             "\n"
5820566             "PORT port number listening for "
5820567             "connections"
5820568             "\n"
5820569             "WWW_ROOT_PATH root for the published "
5820570             "documents." "\n");
5820571 }
```

96.1.18 applic/init.c

Si veda la sezione [92.4](#).

```
5830001 #include <unistd.h>
5830002 #include <stdio.h>
5830003 #include <stdlib.h>
5830004 #include <signal.h>
5830005 #include <sys/wait.h>
5830006 #include <limits.h>
5830007 #include <sys/os32.h>
5830008 #include <fcntl.h>
5830009 #include <string.h>
5830010 //-----
5830011 #define RESPAWN_MAX      7
5830012 #define COMMAND_MAX     100
5830013 #define ARGUMENTS_MAX   32
5830014 #define LINE_MAX        1024
5830015 //-----
5830016 int
5830017 main (int argc, char *argv[], char *envp[])
5830018 {
```

```
5830019 //
5830020 // 'init.c' has its own 'init.crt0.s' with a very
5830021 // small stack
5830022 // size. Remember to verify to have enough room for
5830023 // the stack.
5830024 //
5830025 pid_t pid;
5830026 int status;
5830027 char *exec_argv[ARGUMENTS_MAX];
5830028 int count;
5830029 char *exec_envp[3];
5830030 char buffer[LINE_MAX];
5830031 int r; // Respawn table index.
5830032 int b; // Buffer index.
5830033 size_t size_read;
5830034 char *inittab_id;
5830035 char *inittab_runlevels;
5830036 char *inittab_action;
5830037 char *inittab_process;
5830038 int eof;
5830039 int fd;
5830040 //
5830041 // It follows a table for commands to be respawn.
5830042 //
5830043 struct
5830044 {
5830045     pid_t pid;
5830046     char command[COMMAND_MAX];
5830047 } respawn[RESPAWN_MAX];
5830048
5830049 // -----
5830050 signal (SIGHUP, SIG_IGN);
5830051 signal (SIGINT, SIG_IGN);
5830052 signal (SIGQUIT, SIG_IGN);
5830053 signal (SIGILL, SIG_IGN);
5830054 signal (SIGABRT, SIG_IGN);
5830055 signal (SIGFPE, SIG_IGN);
```

```
5830056 // signal (SIGKILL, SIG_IGN); Cannot ignore SIGKILL.
5830057     signal (SIGSEGV, SIG_IGN);
5830058     signal (SIGPIPE, SIG_IGN);
5830059     signal (SIGALRM, SIG_IGN);
5830060     signal (SIGTERM, SIG_IGN);
5830061 // signal (SIGSTOP, SIG_IGN); Cannot ignore SIGSTOP.
5830062     signal (SIGTSTP, SIG_IGN);
5830063     signal (SIGCONT, SIG_IGN);
5830064     signal (SIGTTIN, SIG_IGN);
5830065     signal (SIGTTOU, SIG_IGN);
5830066     signal (SIGUSR1, SIG_IGN);
5830067     signal (SIGUSR2, SIG_IGN);
5830068     // -----
5830069     printf ("init\n");
5830070     // heap_clear ();
5830071     // process_info ();
5830072     // -----
5830073     //
5830074     // Reset the 'respawn' table.
5830075     //
5830076     for (r = 0; r < RESPAWN_MAX; r++)
5830077     {
5830078         respawn[r].pid = 0;
5830079         respawn[r].command[0] = 0;
5830080         respawn[r].command[COMMAND_MAX - 1] = 0;
5830081     }
5830082     //
5830083     // Read the '/etc/inittab' file.
5830084     //
5830085     fd = open ("/etc/inittab", O_RDONLY);
5830086     //
5830087     if (fd < 0)
5830088     {
5830089         perror ("Cannot open file '/etc/inittab'");
5830090         exit (-1);
5830091     }
5830092     //
```

```
5830093 //
5830094 //
5830095 for (eof = 0, r = 0; !eof && r < RESPAWN_MAX; r++)
5830096 {
5830097     for (b = 0; b < LINE_MAX; b++)
5830098     {
5830099         size_read = read (fd, &buffer[b], (size_t) 1);
5830100         if (size_read <= 0)
5830101         {
5830102             buffer[b] = 0;
5830103             eof = 1; // Close the read loop.
5830104             break;
5830105         }
5830106         if (buffer[b] == '\n')
5830107         {
5830108             buffer[b] = 0;
5830109             break;
5830110         }
5830111     }
5830112 //
5830113 // Remove comments: just replace '#' with '\0'.
5830114 //
5830115 for (b = 0; b < LINE_MAX; b++)
5830116 {
5830117     if (buffer[b] == '#')
5830118     {
5830119         buffer[b] = 0;
5830120         break;
5830121     }
5830122 }
5830123 //
5830124 // If the buffer is an empty string, just loop
5830125 // to next
5830126 // record.
5830127 //
5830128 if (strlen (buffer) == 0)
5830129 {
```



```
5830130         r--;
5830131         continue;
5830132     }
5830133     //
5830134     //
5830135     //
5830136     inittab_id = strtok (buffer, ":");
5830137     inittab_runlevels = strtok (NULL, ":");
5830138     inittab_action = strtok (NULL, ":");
5830139     inittab_process = strtok (NULL, ":");
5830140     //
5830141     // Only action 'respawn' is used.
5830142     //
5830143     if (strcmp (inittab_action, "respawn") == 0)
5830144     {
5830145         strncpy (respawn[r].command, inittab_process,
5830146                 COMMAND_MAX);
5830147     }
5830148     else
5830149     {
5830150         r--;
5830151     }
5830152 }
5830153 //
5830154 //
5830155 //
5830156 close (fd);
5830157 //
5830158 // Define common environment.
5830159 //
5830160 exec_envp[0] = "PATH=/bin:/usr/bin:/sbin:/usr/sbin";
5830161 exec_envp[1] = "CONSOLE=/dev/console";
5830162 exec_envp[2] = NULL;
5830163 //
5830164 // Start processes.
5830165 //
5830166 for (r = 0; r < RESPAWN_MAX; r++)
```

```
5830167     {
5830168         if (strlen (respawn[r].command) > 0)
5830169             {
5830170                 respawn[r].pid = fork ();
5830171                 if (respawn[r].pid == 0)
5830172                     {
5830173                         exec_argv[0] =
5830174                             strtok (respawn[r].command, " \t");
5830175                         for (count = 1;
5830176                             count < (ARGUMENTS_MAX - 2); count++)
5830177                             {
5830178                                 exec_argv[count] = strtok (NULL, " \t");
5830179                                 if (exec_argv[count] == NULL)
5830180                                     {
5830181                                         break;
5830182                                     }
5830183                             }
5830184                         //
5830185                         // Last element must be NULL, even if
5830186                         // there are more
5830187                         // arguments than allowed.
5830188                         //
5830189                         exec_argv[count] = NULL;
5830190                         //
5830191                         // Run!
5830192                         //
5830193                         execve (exec_argv[0], exec_argv, exec_envp);
5830194                         perror (NULL);
5830195                         exit (0);
5830196                     }
5830197             }
5830198     }
5830199     //
5830200     // Wait for the death of child.
5830201     //
5830202     while (1)
5830203         {
```

```
5830204     pid = wait (&status);
5830205     for (r = 0; r < RESPAWN_MAX; r++)
5830206     {
5830207         if (pid == respawn[r].pid)
5830208         {
5830209             //
5830210             // Run it again.
5830211             //
5830212             respawn[r].pid = fork ();
5830213             if (respawn[r].pid == 0)
5830214             {
5830215                 exec_argv[0] =
5830216                     strtok (respawn[r].command, " \t");
5830217                 for (count = 1;
5830218                     count < (ARGUMENTS_MAX - 2); count++)
5830219                 {
5830220                     exec_argv[count] =
5830221                         strtok (NULL, " \t");
5830222                     if (exec_argv[count] == NULL)
5830223                     {
5830224                         break;
5830225                     }
5830226                 }
5830227                 //
5830228                 // Last element must be NULL, even
5830229                 // if there are more
5830230                 // arguments than allowed.
5830231                 //
5830232                 exec_argv[count] = NULL;
5830233                 //
5830234                 // Run!
5830235                 //
5830236                 execve (exec_argv[0], exec_argv,
5830237                       exec_envp);
5830238                 exit (0);
5830239             }
5830240             break;
```

```
5830241     }
5830242     }
5830243 }
5830244 }
```

96.1.19 applic/ipconfig.c

<<

Si veda la sezione [92.5](#).

```
5840001 #include <sys/os32.h>
5840002 #include <kernel/net.h>
5840003 #include <unistd.h>
5840004 #include <stdio.h>
5840005 #include <fcntl.h>
5840006 #include <unistd.h>
5840007 #include <stdlib.h>
5840008 //-----
5840009 #define NET_BUFFER_MAX 1024 // [1]
5840010 //
5840011 // [1] Enough to be able to read important data from
5840012 // the 'net_table[]', without stack overflow.
5840013 // In fact, the table 'net_table[]' contains
5840014 // also the interface frames, and there is no sense
5840015 // to read a full item.
5840016 //
5840017 //-----
5840018 int
5840019 main (int argc, char *argv[], char *envp[])
5840020 {
5840021     int fd;
5840022     ssize_t size_read;
5840023     char buffer[NET_BUFFER_MAX];
5840024     int n;
5840025     net_t *net_table_item;
5840026     char string[80];
5840027
5840028     //
```

```
5840029 // All options are ignored, at the moment.
5840030 //
5840031
5840032 //
5840033 // Open '/dev/kmem_net', to get the network
5840034 // interface table.
5840035 //
5840036 fd = open ("/dev/kmem_net", O_RDONLY);
5840037 if (fd < 0)
5840038 {
5840039     printf ("%s] Cannot open \"/dev/kmem_net\" ",
5840040             argv[0]);
5840041     perror (NULL);
5840042     exit (0);
5840043 }
5840044 //
5840045 // Print header.
5840046 //
5840047 printf ("dev      "
5840048         "address/mask      "
5840049         "mac                " "io      irq\n");
5840050 //
5840051 // Scan NET items and then print body.
5840052 //
5840053 for (n = 0; n < NET_MAX_DEVICES; n++)
5840054 {
5840055     lseek (fd, (off_t) n, SEEK_SET);
5840056     size_read = read (fd, buffer, NET_BUFFER_MAX);
5840057     if (size_read < NET_BUFFER_MAX)
5840058     {
5840059         printf
5840060             ("%s] Cannot read \"/dev/kmem_net\" "
5840061              "item %i ", argv[0], n);
5840062         perror (NULL);
5840063         continue;
5840064     }
5840065     net_table_item = (net_t *) buffer;
```

```
5840066     if (net_table_item->type != NET_DEV_NULL)
5840067     {
5840068         sprintf (string, "net%i      ", n);
5840069         string[6] = '\\0';
5840070         printf ("%s", string);
5840071         //
5840072         sprintf (string, "%i.%i.%i.%i/%i "
5840073                 "
5840074                 ",
5840075                 net_table_item->ip >> 24 & 0x000000FF,
5840076                 net_table_item->ip >> 16 & 0x000000FF,
5840077                 net_table_item->ip >> 8 & 0x000000FF,
5840078                 net_table_item->ip >> 0 & 0x000000FF,
5840079                 net_table_item->m);
5840080         string[20] = '\\0';
5840081         printf ("%s", string);
5840082         //
5840083         if (net_table_item->type & NET_DEV_ETH)
5840084             {
5840085                 printf
5840086                 ("%02x:%02x:%02x:%02x:%02x:%02x  "
5840087                 "0x%04x  %i",
5840088                 net_table_item->ethernet.mac[0],
5840089                 net_table_item->ethernet.mac[1],
5840090                 net_table_item->ethernet.mac[2],
5840091                 net_table_item->ethernet.mac[3],
5840092                 net_table_item->ethernet.mac[4],
5840093                 net_table_item->ethernet.mac[5],
5840094                 net_table_item->ethernet.base_io,
5840095                 net_table_item->ethernet.irq);
5840096             }
5840097         printf ("\n");
5840098     }
5840099     close (fd);
5840100     return (0);
5840101 }
```

96.1.20 applic/kill.c



Si veda la sezione [86.12](#).

```
5850001 #include <sys/os32.h>
5850002 #include <sys/stat.h>
5850003 #include <sys/types.h>
5850004 #include <unistd.h>
5850005 #include <stdlib.h>
5850006 #include <fcntl.h>
5850007 #include <errno.h>
5850008 #include <signal.h>
5850009 #include <stdio.h>
5850010 #include <string.h>
5850011 #include <limits.h>
5850012 #include <libgen.h>
5850013 //-----
5850014 static void usage (void);
5850015 //-----
5850016 int
5850017 main (int argc, char *argv[], char *envp[])
5850018 {
5850019     int signal;
5850020     int pid;
5850021     int a;          // Index inside arguments.
5850022     int option_s = 0;
5850023     int option_l = 0;
5850024     int opt;
5850025     extern char *optarg;
5850026     extern int optopt;
5850027     //
5850028     // There must be at least an option, plus the
5850029     // program name.
5850030     //
5850031     if (argc < 2)
5850032     {
5850033         usage ();
5850034         return (1);
```



```
5850072         {
5850073             signal = SIGQUIT;
5850074         }
5850075     else if (strcmp (optarg, "ILL") == 0)
5850076     {
5850077         signal = SIGILL;
5850078     }
5850079     else if (strcmp (optarg, "ABRT") == 0)
5850080     {
5850081         signal = SIGABRT;
5850082     }
5850083     else if (strcmp (optarg, "FPE") == 0)
5850084     {
5850085         signal = SIGFPE;
5850086     }
5850087     else if (strcmp (optarg, "KILL") == 0)
5850088     {
5850089         signal = SIGKILL;
5850090     }
5850091     else if (strcmp (optarg, "SEGV") == 0)
5850092     {
5850093         signal = SIGSEGV;
5850094     }
5850095     else if (strcmp (optarg, "PIPE") == 0)
5850096     {
5850097         signal = SIGPIPE;
5850098     }
5850099     else if (strcmp (optarg, "ALRM") == 0)
5850100     {
5850101         signal = SIGALRM;
5850102     }
5850103     else if (strcmp (optarg, "TERM") == 0)
5850104     {
5850105         signal = SIGTERM;
5850106     }
5850107     else if (strcmp (optarg, "STOP") == 0)
5850108     {
```

```
5850109         signal = SIGSTOP;
5850110     }
5850111     else if (strcmp (optarg, "TSTP") == 0)
5850112     {
5850113         signal = SIGTSTP;
5850114     }
5850115     else if (strcmp (optarg, "CONT") == 0)
5850116     {
5850117         signal = SIGCONT;
5850118     }
5850119     else if (strcmp (optarg, "CHLD") == 0)
5850120     {
5850121         signal = SIGCHLD;
5850122     }
5850123     else if (strcmp (optarg, "TTIN") == 0)
5850124     {
5850125         signal = SIGTTIN;
5850126     }
5850127     else if (strcmp (optarg, "TTOU") == 0)
5850128     {
5850129         signal = SIGTTOU;
5850130     }
5850131     else if (strcmp (optarg, "USR1") == 0)
5850132     {
5850133         signal = SIGUSR1;
5850134     }
5850135     else if (strcmp (optarg, "USR2") == 0)
5850136     {
5850137         signal = SIGUSR2;
5850138     }
5850139     else
5850140     {
5850141         fprintf (stderr, "Unknown signal %s.\n",
5850142                 optarg);
5850143         return (1);
5850144     }
5850145     break;
```

```
5850146     case '?':
5850147         fprintf (stderr, "Unknown option -%c.\n", optopt);
5850148         usage ();
5850149         return (1);
5850150         break;
5850151     case ':':
5850152         fprintf (stderr,
5850153                 "Missing argument for option -%c\n",
5850154                 optopt);
5850155         usage ();
5850156         return (1);
5850157         break;
5850158     default:
5850159         fprintf (stderr,
5850160                 "Getopt problem: unknown option "
5850161                 "%c\n", opt);
5850162         return (1);
5850163     }
5850164 }
5850165 //
5850166 //
5850167 //
5850168 if (option_l && option_s)
5850169 {
5850170     fprintf (stderr,
5850171             "Options \"-l\" and \"-s\" together ");
5850172     fprintf (stderr, "are incompatible.\n");
5850173     usage ();
5850174     return (1);
5850175 }
5850176 //
5850177 // Option "-l".
5850178 //
5850179 if (option_l)
5850180 {
5850181     printf ("HUP ");
5850182     printf ("INT ");
```

```
5850183     printf ("QUIT ");
5850184     printf ("ILL ");
5850185     printf ("ABRT ");
5850186     printf ("FPE ");
5850187     printf ("KILL ");
5850188     printf ("SEGV ");
5850189     printf ("PIPE ");
5850190     printf ("ALRM ");
5850191     printf ("TERM ");
5850192     printf ("STOP ");
5850193     printf ("TSTP ");
5850194     printf ("CONT ");
5850195     printf ("CHLD ");
5850196     printf ("TTIN ");
5850197     printf ("TTOU ");
5850198     printf ("USR1 ");
5850199     printf ("USR2 ");
5850200     printf ("\n");
5850201 }
5850202 //
5850203 // Option "-s".
5850204 //
5850205 if (option_s)
5850206 {
5850207     //
5850208     // Scan arguments.
5850209     //
5850210     for (a = 3; a < argc; a++)
5850211     {
5850212         //
5850213         // Get PID.
5850214         //
5850215         pid = atoi (argv[a]);
5850216         if (pid > 0)
5850217         {
5850218             //
5850219             // Kill.
```

```
5850220         //
5850221         if (kill (pid, signal) < 0)
5850222             {
5850223                 perror (argv[a]);
5850224             }
5850225         }
5850226     else
5850227     {
5850228         fprintf (stderr, "Invalid PID %s.", argv[a]);
5850229     }
5850230 }
5850231 }
5850232 //
5850233 // All done.
5850234 //
5850235 return (0);
5850236 }
5850237
5850238 //-----
5850239 static void
5850240 usage (void)
5850241 {
5850242     fprintf (stderr, "Usage: kill -s SIGNAL_NAME PID...\n");
5850243     fprintf (stderr, "        kill -l\n");
5850244 }
```

96.1.21 applic/ln.c

Si veda la sezione [86.13](#).

```
5860001 #include <sys/os32.h>
5860002 #include <sys/stat.h>
5860003 #include <sys/types.h>
5860004 #include <unistd.h>
5860005 #include <stdlib.h>
5860006 #include <fcntl.h>
5860007 #include <errno.h>
```



```
5860008 #include <signal.h>
5860009 #include <stdio.h>
5860010 #include <string.h>
5860011 #include <limits.h>
5860012 #include <libgen.h>
5860013 //-----
5860014 static void usage (void);
5860015 //-----
5860016 int
5860017 main (int argc, char *argv[], char *envp[])
5860018 {
5860019     char *source;
5860020     char *destination;
5860021     char *new_destination;
5860022     struct stat file_status;
5860023     int dest_is_a_dir = 0;
5860024     int a;          // Argument index.
5860025     char path[PATH_MAX];
5860026     //
5860027     // There must be at least two arguments, plus the
5860028     // program name.
5860029     //
5860030     if (argc < 3)
5860031     {
5860032         usage ();
5860033         return (1);
5860034     }
5860035     //
5860036     // Select the last argument as the destination.
5860037     //
5860038     destination = argv[argc - 1];
5860039     //
5860040     // Check if it is a directory and save it in a flag.
5860041     //
5860042     if (stat (destination, &file_status) == 0)
5860043     {
5860044         if (S_ISDIR (file_status.st_mode))
```

```
5860045     {
5860046         dest_is_a_dir = 1;
5860047     }
5860048 }
5860049 //
5860050 // If there are more than two arguments, verify that
5860051 // the last
5860052 // one is a directory.
5860053 //
5860054 if (argc > 3)
5860055     {
5860056     if (!dest_is_a_dir)
5860057         {
5860058         usage ();
5860059         fprintf (stderr, "The destination \"%s\" ",
5860060                 destination);
5860061         fprintf (stderr, "is not a directory!\n");
5860062         return (1);
5860063         }
5860064     }
5860065 //
5860066 // Scan the arguments, excluded the last, that is
5860067 // the destination.
5860068 //
5860069 for (a = 1; a < (argc - 1); a++)
5860070     {
5860071     //
5860072     // Source.
5860073     //
5860074     source = argv[a];
5860075     //
5860076     // Verify access permissions.
5860077     //
5860078     if (access (source, R_OK) < 0)
5860079         {
5860080         perror (source);
5860081         continue;
```

```
5860082     }
5860083     //
5860084     // Destination.
5860085     //
5860086     // If it is a directory, the destination path
5860087     // must be corrected.
5860088     //
5860089     if (dest_is_a_dir)
5860090     {
5860091         path[0] = 0;
5860092         strcat (path, destination);
5860093         strcat (path, "/");
5860094         strcat (path, basename (source));
5860095         //
5860096         // Update the destination path.
5860097         //
5860098         new_destination = path;
5860099     }
5860100     else
5860101     {
5860102         new_destination = destination;
5860103     }
5860104     //
5860105     // Check if destination file exists.
5860106     //
5860107     if (stat (new_destination, &file_status) == 0)
5860108     {
5860109         fprintf (stderr,
5860110                 "The destination file, \"%s\", ",
5860111                 new_destination);
5860112         fprintf (stderr, "already exists!\n");
5860113         continue;
5860114     }
5860115     //
5860116     // Everything is ready for the link.
5860117     //
5860118     if (link (source, new_destination) < 0)
```



```
5860119     {
5860120         perror (new_destination);
5860121         continue;
5860122     }
5860123 }
5860124 //
5860125 // All done.
5860126 //
5860127 return (0);
5860128 }
5860129
5860130 //-----
5860131 static void
5860132 usage (void)
5860133 {
5860134     fprintf (stderr, "Usage: ln OLD_NAME NEW_NAME\n");
5860135     fprintf (stderr, "          ln FILE... DIRECTORY\n");
5860136 }
```

96.1.22 applic/login.c

Si veda la sezione [86.14](#).

```
5870001 #include <unistd.h>
5870002 #include <stdlib.h>
5870003 #include <sys/stat.h>
5870004 #include <sys/types.h>
5870005 #include <fcntl.h>
5870006 #include <errno.h>
5870007 #include <unistd.h>
5870008 #include <signal.h>
5870009 #include <stdio.h>
5870010 #include <sys/wait.h>
5870011 #include <stdio.h>
5870012 #include <string.h>
5870013 #include <limits.h>
5870014 #include <stdint.h>
```



```
5870015 #include <sys/os32.h>
5870016 //-----
5870017 #define LOGIN_MAX      64
5870018 #define PASSWORD_MAX   64
5870019 #define HOME_MAX       64
5870020 #define LINE_MAX       1024
5870021 //-----
5870022 int
5870023 main (int argc, char *argv[], char *envp[])
5870024 {
5870025     char login[LOGIN_MAX];
5870026     char password[PASSWORD_MAX];
5870027     char buffer[LINE_MAX];
5870028     char *user_name;
5870029     char *user_password;
5870030     char *user_uid;
5870031     char *user_gid;
5870032     char *user_description;
5870033     char *user_home;
5870034     char *user_shell;
5870035     uid_t uid;
5870036     uid_t euid;
5870037     gid_t gid;
5870038     gid_t egid;
5870039     int fd;
5870040     ssize_t size_read;
5870041     int b;           // Index inside buffer.
5870042     int loop;
5870043     char *exec_argv[2];
5870044     int status;
5870045     char *tty_path;
5870046     //
5870047     // Check if login is running correctly.
5870048     //
5870049     euid = geteuid ();
5870050     uid = getuid ();
5870051     //
```

```
5870052 // Check privileges.
5870053 //
5870054 if (!(uid == 0 && euid == 0))
5870055 {
5870056     printf
5870057     ("%s: can only run with root privileges!\n",
5870058     argv[0]);
5870059     exit (-1);
5870060 }
5870061 //
5870062 // Prepare arguments for the shell call.
5870063 //
5870064 exec_argv[0] = "-";
5870065 exec_argv[1] = NULL;
5870066 //
5870067 // Login.
5870068 //
5870069 while (1)
5870070 {
5870071     fd = open ("/etc/passwd", O_RDONLY);
5870072     //
5870073     if (fd < 0)
5870074     {
5870075         perror ("Cannot open file '/etc/passwd'");
5870076         exit (-1);
5870077     }
5870078     //
5870079     printf ("Log in as \"root\" or \"user\" "
5870080            "with password \"ciao\" :-)\n");
5870081     input_line (login, "login: ", LOGIN_MAX,
5870082               INPUT_LINE_ECHO);
5870083     //
5870084     //
5870085     //
5870086     loop = 1;
5870087     while (loop)
5870088     {
```

```
5870089     for (b = 0; b < LINE_MAX; b++)
5870090     {
5870091         size_read = read (fd, &buffer[b], (size_t) 1);
5870092         if (size_read <= 0)
5870093         {
5870094             buffer[b] = 0;
5870095             loop = 0;      // Close the middle
5870096                          // loop.
5870097             break;
5870098         }
5870099         if (buffer[b] == '\n')
5870100         {
5870101             buffer[b] = 0;
5870102             break;
5870103         }
5870104     }
5870105     //
5870106     // Please notice that 'strtok()' does not
5870107     // allow to have empty fields! If it finds
5870108     // a ':::', it will treat it as a single ':'.
5870109     //
5870110     user_name = strtok (buffer, ":");
5870111     user_password = strtok (NULL, ":");
5870112     user_uid = strtok (NULL, ":");
5870113     user_gid = strtok (NULL, ":");
5870114     user_description = strtok (NULL, ":");
5870115     user_home = strtok (NULL, ":");
5870116     user_shell = strtok (NULL, ":");
5870117     //
5870118     if (strcmp (user_name, login) == 0)
5870119     {
5870120         input_line (password, "password: ",
5870121                   PASSWORD_MAX, INPUT_LINE_HIDDEN);
5870122         //
5870123         // Compare passwords: empty passwords
5870124         // are not allowed.
5870125         //
```

```
5870126         if (strcmp (user_password, password) == 0)
5870127             {
5870128                 uid = atoi (user_uid);
5870129                 euid = uid;
5870130                 gid = atoi (user_gid);
5870131                 egid = gid;
5870132                 //
5870133                 // Find the controlling terminal and
5870134                 // change
5870135                 // property and access permissions.
5870136                 //
5870137                 tty_path = ttyname (STDIN_FILENO);
5870138                 if (tty_path != NULL)
5870139                     {
5870140                         status = chown (tty_path, uid, 0);
5870141                         if (status != 0)
5870142                             {
5870143                                 perror (NULL);
5870144                             }
5870145                         status = chmod (tty_path, 0600);
5870146                         if (status != 0)
5870147                             {
5870148                                 perror (NULL);
5870149                             }
5870150                     }
5870151                 //
5870152                 // Cd to the home directory, if
5870153                 // present.
5870154                 //
5870155                 status = chdir (user_home);
5870156                 if (status != 0)
5870157                     {
5870158                         perror (NULL);
5870159                     }
5870160                 //
5870161                 // Now change personality: first the
5870162                 // group,
```

```
5870163 // otherwise, it would be not
5870164 // possible to do
5870165 // after changing the UID to an
5870166 // unprivileged
5870167 // one.
5870168 //
5870169 setgid (gid);
5870170 setegid (egid);
5870171 //
5870172 setuid (uid);
5870173 seteuid (euid);
5870174 //
5870175 // Run the shell, replacing the
5870176 // login process; the
5870177 // environment is taken from 'init'.
5870178 //
5870179 execve (user_shell, exec_argv, envp);
5870180 exit (0);
5870181 }
5870182 //
5870183 // Login failed: will try again.
5870184 //
5870185 loop = 0; // Close the middle loop.
5870186 break;
5870187 }
5870188 }
5870189 close (fd);
5870190 }
5870191 }
```

96.1.23 applic/ls.c



Si veda la sezione [86.15](#).

```
5880001 #include <sys/stat.h>
5880002 #include <sys/types.h>
5880003 #include <unistd.h>
```

```
5880004 #include <stdlib.h>
5880005 #include <fcntl.h>
5880006 #include <errno.h>
5880007 #include <signal.h>
5880008 #include <stdio.h>
5880009 #include <string.h>
5880010 #include <limits.h>
5880011 #include <libgen.h>
5880012 #include <dirent.h>
5880013 #include <pwd.h>
5880014 #include <grp.h>
5880015 #include <time.h>
5880016 //-----
5880017 #define BUFFER_SIZE      131072
5880018 #define LIST_SIZE        8000
5880019 //-----
5880020 static void usage (void);
5880021 static int compare (const void *p1, const void *p2);
5880022 //-----
5880023 //
5880024 // Static variables to avoid stack overflow.
5880025 //
5880026 static char buffer[BUFFER_SIZE];
5880027 static char *list[LIST_SIZE];
5880028 //-----
5880029 int
5880030 main (int argc, char *argv[], char *envp[])
5880031 {
5880032     int option_a = 0;
5880033     int option_l = 0;
5880034     int opt;
5880035     // extern char *optarg; // not used.
5880036     extern int optind;
5880037     extern int optopt;
5880038     struct stat file_status;
5880039     DIR *dp;
5880040     struct dirent *dir;
```

```
5880041 int b;           // Buffer index.
5880042 int l;           // List index.
5880043 int len;        // Name length.
5880044 char *path = NULL;
5880045 char pathname[PATH_MAX];
5880046 struct passwd *pws;
5880047 struct group *grs;
5880048 struct tm *tms;
5880049 //
5880050 // Check for options.
5880051 //
5880052 while ((opt = getopt (argc, argv, ":al")) != -1)
5880053 {
5880054     switch (opt)
5880055     {
5880056     case 'l':
5880057         option_l = 1;
5880058         break;
5880059     case 'a':
5880060         option_a = 1;
5880061         break;
5880062     case '?':
5880063         fprintf (stderr, "Unknown option -%c.\n", optopt);
5880064         usage ();
5880065         return (1);
5880066         break;
5880067     case ':':
5880068         fprintf (stderr,
5880069                 "Missing argument for option -%c\n",
5880070                 optopt);
5880071         usage ();
5880072         return (1);
5880073         break;
5880074     default:
5880075         fprintf (stderr,
5880076                 "Getopt problem: unknown option "
5880077                 "%c\n", opt);
```



```
5880078         return (1);
5880079     }
5880080 }
5880081 //
5880082 // If no arguments are present, at least the current
5880083 // directory is
5880084 // read.
5880085 //
5880086 if (optind == argc)
5880087 {
5880088     //
5880089     // There are no more arguments. Replace the
5880090     // program name,
5880091     // corresponding to 'argv[0]', with the current
5880092     // directory
5880093     // path string.
5880094     //
5880095     argv[0] = ".";
5880096     argc = 1;
5880097     optind = 0;
5880098 }
5880099 //
5880100 // This is a very simplified 'ls': if there is only
5880101 // a name
5880102 // and it is a directory, the directory content is
5880103 // taken as
5880104 // the new 'argv[]' array.
5880105 //
5880106 if (optind == (argc - 1))
5880107 {
5880108     //
5880109     // There is a request for a single name. Test if
5880110     // it exists
5880111     // and if it is a directory.
5880112     //
5880113     if (stat (argv[optind], &file_status) != 0)
5880114     {
```

```
5880115     fprintf (stderr,
5880116             "File \"%s\" does not exist!\n",
5880117             argv[optind]);
5880118     return (2);
5880119 }
5880120 //
5880121 if (S_ISDIR (file_status.st_mode))
5880122 {
5880123     //
5880124     // Save the directory inside the 'path'
5880125     // pointer.
5880126     //
5880127     path = argv[optind];
5880128     //
5880129     // Open the directory.
5880130     //
5880131     dp = opendir (argv[optind]);
5880132     if (dp == NULL)
5880133     {
5880134         perror (argv[optind]);
5880135         return (3);
5880136     }
5880137     //
5880138     // Read the directory and fill the buffer
5880139     // with names.
5880140     //
5880141     b = 0;
5880142     l = 0;
5880143     while ((dir = readdir (dp)) != NULL)
5880144     {
5880145         len = strlen (dir->d_name);
5880146         //
5880147         // Check if the buffer can hold it.
5880148         //
5880149         if ((b + len + 1) > BUFFER_SIZE)
5880150         {
5880151             fprintf (stderr, "not enough memory\n");
```

```
5880152         break;
5880153     }
5880154     //
5880155     // Consider the directory item only if
5880156     // there is
5880157     // a valid name. If it is empty, just
5880158     // ignore it.
5880159     //
5880160     if (len > 0)
5880161     {
5880162         strcpy (&buffer[b], dir->d_name);
5880163         list[l] = &buffer[b];
5880164         b += len + 1;
5880165         l++;
5880166     }
5880167 }
5880168 //
5880169 // Close the directory.
5880170 //
5880171 closedir (dp);
5880172 //
5880173 // Sort the list.
5880174 //
5880175 qsort (list, (size_t) l, sizeof (char *),
5880176       compare);
5880177 //
5880178 // Convert the directory list into a new
5880179 // 'argv[]' array,
5880180 // with a valid 'argc'. The variable
5880181 // 'optind' must be
5880182 // reset to the first element index, because
5880183 // there is
5880184 // no program name inside the new 'argv[]'
5880185 // at index zero.
5880186 //
5880187 argv = list;
5880188 argc = l;
```

```
5880189         optind = 0;
5880190     }
5880191 }
5880192 //
5880193 // Scan arguments, or list converted into 'argv[]'.
5880194 //
5880195 for (; optind < argc; optind++)
5880196 {
5880197     if (argv[optind][0] == '.')
5880198     {
5880199         //
5880200         // Current name starts with '.'.
5880201         //
5880202         if (!option_a)
5880203         {
5880204             //
5880205             // Do not show name starting with '.'.
5880206             //
5880207             continue;
5880208         }
5880209     }
5880210 //
5880211 // Build the pathname.
5880212 //
5880213 if (path == NULL)
5880214 {
5880215     strcpy (&pathname[0], argv[optind]);
5880216 }
5880217 else
5880218 {
5880219     strcpy (pathname, path);
5880220     strcat (pathname, "/");
5880221     strcat (pathname, argv[optind]);
5880222 }
5880223 //
5880224 // Check if file exists, reading status.
5880225 //
```

```
5880226     if (stat (pathname, &file_status) != 0)
5880227     {
5880228         fprintf (stderr,
5880229             "File \"%s\" does not exist!\n",
5880230             pathname);
5880231         return (2);
5880232     }
5880233     //
5880234     // Show file name.
5880235     //
5880236     if (option_1)
5880237     {
5880238         //
5880239         // Print the file type.
5880240         //
5880241         if (S_ISBLK (file_status.st_mode))
5880242             printf ("b");
5880243         else if (S_ISCHR (file_status.st_mode))
5880244             printf ("c");
5880245         else if (S_ISFIFO (file_status.st_mode))
5880246             printf ("p");
5880247         else if (S_ISREG (file_status.st_mode))
5880248             printf ("-");
5880249         else if (S_ISDIR (file_status.st_mode))
5880250             printf ("d");
5880251         else if (S_ISLNK (file_status.st_mode))
5880252             printf ("l");
5880253         else if (S_ISSOCK (file_status.st_mode))
5880254             printf ("s");
5880255         else
5880256             printf ("?");
5880257         //
5880258         // Print permissions.
5880259         //
5880260         if (S_IRUSR & file_status.st_mode)
5880261             printf ("r");
5880262         else
```

```
5880263     printf ("-");
5880264     if (S_IWUSR & file_status.st_mode)
5880265         printf ("w");
5880266     else
5880267         printf ("-");
5880268     if (S_IXUSR & file_status.st_mode)
5880269         printf ("x");
5880270     else
5880271         printf ("-");
5880272     if (S_IRGRP & file_status.st_mode)
5880273         printf ("r");
5880274     else
5880275         printf ("-");
5880276     if (S_IWGRP & file_status.st_mode)
5880277         printf ("w");
5880278     else
5880279         printf ("-");
5880280     if (S_IXGRP & file_status.st_mode)
5880281         printf ("x");
5880282     else
5880283         printf ("-");
5880284     if (S_IROTH & file_status.st_mode)
5880285         printf ("r");
5880286     else
5880287         printf ("-");
5880288     if (S_IWOTH & file_status.st_mode)
5880289         printf ("w");
5880290     else
5880291         printf ("-");
5880292     if (S_IXOTH & file_status.st_mode)
5880293         printf ("x");
5880294     else
5880295         printf ("-");
5880296     //
5880297     // Print links.
5880298     //
5880299     printf (" %3i", (int) file_status.st_nlink);
```

```
5880300 //
5880301 // Print owner.
5880302 //
5880303 pws = getpwuid (file_status.st_uid);
5880304 if (pws == NULL)
5880305     {
5880306         printf (" %i", (int) file_status.st_uid);
5880307     }
5880308 else
5880309     {
5880310         printf (" %s", pws->pw_name);
5880311     }
5880312 //
5880313 // Print group.
5880314 //
5880315 grs = getgrgid (file_status.st_gid);
5880316 if (grs == NULL)
5880317     {
5880318         printf (" %i", (int) file_status.st_gid);
5880319     }
5880320 else
5880321     {
5880322         printf (" %s", grs->gr_name);
5880323     }
5880324 //
5880325 // Print file size or device major-minor.
5880326 //
5880327 if (S_ISBLK (file_status.st_mode)
5880328     || S_ISCHR (file_status.st_mode))
5880329     {
5880330         printf (" %3i,",
5880331             (int) major (file_status.st_rdev));
5880332         printf (" %3i",
5880333             (int) minor (file_status.st_rdev));
5880334     }
5880335 else
5880336     {
```

```
5880337         printf (" %8i", (int) file_status.st_size);
5880338     }
5880339     //
5880340     // Print modification date and time.
5880341     //
5880342     tms = localtime (&(file_status.st_mtime));
5880343     printf (" %4u-%02u-%02u %02u:%02u",
5880344             tms->tm_year, tms->tm_mon,
5880345             tms->tm_mday, tms->tm_hour, tms->tm_min);
5880346     //
5880347     // Print file name, but with no additional
5880348     // path.
5880349     //
5880350     printf (" %s\n", argv[optind]);
5880351 }
5880352 else
5880353 {
5880354     //
5880355     // Just show the file name and go to the
5880356     // next line.
5880357     //
5880358     printf ("%s\n", argv[optind]);
5880359 }
5880360 }
5880361 //
5880362 // All done.
5880363 //
5880364 return (0);
5880365 }
5880366
5880367 //-----
5880368 static void
5880369 usage (void)
5880370 {
5880371     fprintf (stderr, "Usage: ls [OPTION] [FILE]...\n");
5880372 }
5880373
```



```
5880374 //-----
5880375 static int
5880376 compare (const void *p1, const void *p2)
5880377 {
5880378     return (strcmp (*(char **) p1), (*(char **) p2));
5880379 }
```

96.1.24 applic/man.c

Si veda la sezione [86.16](#).

```
5890001 #include <unistd.h>
5890002 #include <stdlib.h>
5890003 #include <errno.h>
5890004 //-----
5890005 #define MAX_LINES    20
5890006 #define MAX_COLUMNS  80
5890007 //-----
5890008 static char *man_page_directory = "/usr/share/man";
5890009 //-----
5890010 static void usage (void);
5890011 static FILE *open_man_page (int section, char *name);
5890012 static void build_path_name (int section, char *name,
5890013                             char *path);
5890014 //-----
5890015 int
5890016 main (int argc, char *argv[], char *envp[])
5890017 {
5890018     FILE *fp;
5890019     char *name;
5890020     int section;
5890021     int c;
5890022     int line = 1; // Line internal counter.
5890023     int column = 1; // Column internal counter.
5890024     int loop;
5890025     //
5890026     // There must be minimum an argument, and maximum
```

```
5890027 // two.
5890028 //
5890029 if (argc < 2 || argc > 3)
5890030 {
5890031     usage ();
5890032     return (1);
5890033 }
5890034 //
5890035 // If there are two arguments, there must be the
5890036 // section number.
5890037 //
5890038 if (argc == 3)
5890039 {
5890040     section = atoi (argv[1]);
5890041     name = argv[2];
5890042 }
5890043 else
5890044 {
5890045     section = 0;
5890046     name = argv[1];
5890047 }
5890048 //
5890049 // Try to open the manual page.
5890050 //
5890051 fp = open_man_page (section, name);
5890052 //
5890053 if (fp == NULL)
5890054 {
5890055     //
5890056     // Error opening file.
5890057     //
5890058     return (1);
5890059 }
5890060
5890061 //
5890062 // The following loop continues while the file
5890063 // gives characters, or when a command to change
```

```
5890064 // file or to quit is given.
5890065 //
5890066 for (loop = 1; loop;)
5890067 {
5890068 //
5890069 // Read a single character.
5890070 //
5890071 c = getc (fp);
5890072 //
5890073 if (c == EOF)
5890074 {
5890075     loop = 0;
5890076     break;
5890077 }
5890078 //
5890079 // If the character read is a special one,
5890080 // the line/column calculation is modified,
5890081 // so that it is known when to stop scrolling.
5890082 //
5890083 switch (c)
5890084 {
5890085     case '\r':
5890086         //
5890087         // Displaying this character, the cursor
5890088         // should go
5890089         // back to the first column. So the column
5890090         // counter
5890091         // is reset.
5890092         //
5890093         column = 1;
5890094         break;
5890095     case '\n':
5890096         //
5890097         // Displaying this character, the cursor
5890098         // should go
5890099         // back to the next line, at the first
5890100         // column.
```

```
5890101 // So the column counter is reset and the
5890102 // line
5890103 // counter is incremented.
5890104 //
5890105 line++;
5890106 column = 1;
5890107 break;
5890108 case '\b':
5890109 //
5890110 // Displaying this character, the cursor
5890111 // should go
5890112 // back one position, unless it is already
5890113 // at the
5890114 // beginning.
5890115 //
5890116 if (column > 1)
5890117 {
5890118     column--;
5890119 }
5890120 break;
5890121 default:
5890122 //
5890123 // Any other character must increase the
5890124 // column
5890125 // counter.
5890126 //
5890127 column++;
5890128 }
5890129 //
5890130 // Display the character, even if it is a
5890131 // special one:
5890132 // it is responsibility of the screen device
5890133 // management
5890134 // to do something good with special characters.
5890135 //
5890136 putchar (c);
5890137 //
```

```
5890138 // If the column counter is gone beyond the
5890139 // screen columns,
5890140 // then adjust the column counter and increment
5890141 // the line
5890142 // counter.
5890143 //
5890144 if (column > MAX_COLUMNS)
5890145 {
5890146     column -= MAX_COLUMNS;
5890147     line++;
5890148 }
5890149 //
5890150 // Check if there is space for scrolling.
5890151 //
5890152 if (line < MAX_LINES)
5890153 {
5890154     continue;
5890155 }
5890156 //
5890157 // Here, displayed lines are MAX_LINES.
5890158 //
5890159 if (column > 1)
5890160 {
5890161     //
5890162     // Something was printed at the current
5890163     // line: must
5890164     // do a new line.
5890165     //
5890166     putchar ('\n');
5890167 }
5890168 //
5890169 // Show the more prompt.
5890170 //
5890171 printf ("--More--");
5890172 fflush (stdout);
5890173 //
5890174 // Read a character from standard input.
```

```
5890175 //
5890176 c = getchar ();
5890177 //
5890178 // Consider command 'q', but any other character
5890179 // can be introduced, to let show the next page.
5890180 //
5890181 switch (c)
5890182 {
5890183 case 'Q':
5890184 case 'q':
5890185 //
5890186 // Quit. But must erase the '--More--'
5890187 // prompt.
5890188 //
5890189 printf ("\b \b\b \b\b \b\b \b\b \b");
5890190 printf ("\b \b\b \b\b \b\b \b");
5890191 fclose (fp);
5890192 return (0);
5890193 }
5890194 //
5890195 // Backspace to overwrite '--More--' and the
5890196 // character
5890197 // pressed.
5890198 //
5890199 printf
5890200 (" \b \b\b \b\b \b\b \b\b "
5890201 "\b\b \b\b \b\b \b\b \b");
5890202 //
5890203 // Reset line/column counters.
5890204 //
5890205 column = 1;
5890206 line = 1;
5890207 }
5890208 //
5890209 // Close the file pointer if it is still open.
5890210 //
5890211 if (fp != NULL)
```

```
5890212     {
5890213         fclose (fp);
5890214     }
5890215     //
5890216     return (0);
5890217 }
5890218
5890219 //-----
5890220 static void
5890221 usage (void)
5890222 {
5890223     fprintf (stderr, "Usage: man [SECTION] NAME\n");
5890224 }
5890225
5890226 //-----
5890227 FILE *
5890228 open_man_page (int section, char *name)
5890229 {
5890230     FILE *fp;
5890231     char path[PATH_MAX];
5890232     struct stat file_status;
5890233     //
5890234     //
5890235     //
5890236     if (section > 0)
5890237     {
5890238         build_path_name (section, name, path);
5890239         //
5890240         // Check if file exists.
5890241         //
5890242         if (stat (path, &file_status) != 0)
5890243         {
5890244             fprintf (stderr,
5890245                     "Man page %s(%i) does not exist!\n",
5890246                     name, section);
5890247             return (NULL);
5890248         }

```

```
5890249     }
5890250 else
5890251     {
5890252         //
5890253         // Must try a section.
5890254         //
5890255         for (section = 1; section < 9; section++)
5890256             {
5890257                 build_path_name (section, name, path);
5890258                 //
5890259                 // Check if file exists.
5890260                 //
5890261                 if (stat (path, &file_status) == 0)
5890262                     {
5890263                         //
5890264                         // Found.
5890265                         //
5890266                         break;
5890267                     }
5890268             }
5890269     }
5890270 //
5890271 // Check if a file was found.
5890272 //
5890273 if (section < 9)
5890274     {
5890275         fp = fopen (path, "r");
5890276         //
5890277         if (fp == NULL)
5890278             {
5890279                 //
5890280                 // Error opening file.
5890281                 //
5890282                 perror (path);
5890283                 return (NULL);
5890284             }
5890285     else
```



```
5890286         {
5890287             //
5890288             // Opened right.
5890289             //
5890290             return (fp);
5890291         }
5890292     }
5890293     else
5890294     {
5890295         fprintf (stderr, "Man page %s does not exist!\n",
5890296                 name);
5890297         return (NULL);
5890298     }
5890299 }
5890300
5890301 //-----
5890302 void
5890303 build_path_name (int section, char *name, char *path)
5890304 {
5890305     char string_section[10];
5890306     //
5890307     // Convert the section number into a string.
5890308     //
5890309     sprintf (string_section, "%i", section);
5890310     //
5890311     // Prepare the path to the man file.
5890312     //
5890313     path[0] = 0;
5890314     strcat (path, man_page_directory);
5890315     strcat (path, "/");
5890316     strcat (path, name);
5890317     strcat (path, ".");
5890318     strcat (path, string_section);
5890319 }
```

96.1.25 applic/mkdir.c



Si veda la sezione [86.17](#).

```
5900001 #include <sys/os32.h>
5900002 #include <sys/stat.h>
5900003 #include <sys/types.h>
5900004 #include <unistd.h>
5900005 #include <stdlib.h>
5900006 #include <fcntl.h>
5900007 #include <errno.h>
5900008 #include <signal.h>
5900009 #include <stdio.h>
5900010 #include <string.h>
5900011 #include <limits.h>
5900012 #include <libgen.h>
5900013 //-----
5900014 static int mkdir_parents (const char *path, mode_t mode);
5900015 static void usage (void);
5900016 //-----
5900017 int
5900018 main (int argc, char *argv[], char *envp[])
5900019 {
5900020     sysmsg_uarea_t msg;
5900021     int status;
5900022     mode_t mode = 0;
5900023     int m;          // Index inside mode argument.
5900024     int digit;
5900025     char **dir;
5900026     int d;          // Directory index.
5900027     int option_p = 0;
5900028     int option_m = 0;
5900029     int opt;
5900030     extern char *optarg;
5900031     extern int optind;
5900032     extern int optopt;
5900033     //
5900034     // There must be at least an argument, plus the
```

```
5900035 // program name.
5900036 //
5900037 if (argc < 2)
5900038 {
5900039     usage ();
5900040     return (1);
5900041 }
5900042 //
5900043 // Check for options, starting from 'p'. The 'dir'
5900044 // pointer is used
5900045 // to calculate the argument pointer to the first
5900046 // directory [1].
5900047 // The macroinstruction 'max()' is declared inside
5900048 // <sys/os32.h>
5900049 // and does the expected thing.
5900050 //
5900051 while ((opt = getopt (argc, argv, ":pm:")) != -1)
5900052 {
5900053     switch (opt)
5900054     {
5900055     case 'm':
5900056         option_m = 1;
5900057         for (m = 0; m < strlen (optarg); m++)
5900058         {
5900059             digit = (optarg[m] - '0');
5900060             if (digit < 0 || digit > 7)
5900061             {
5900062                 usage ();
5900063                 return (2);
5900064             }
5900065             mode = mode * 8 + digit;
5900066         }
5900067         break;
5900068     case 'p':
5900069         option_p = 1;
5900070         break;
5900071     case '?':
```

```
5900072     printf ("Unknown option -%c.\n", optopt);
5900073     usage ();
5900074     return (1);
5900075     break;
5900076     case ':':
5900077         printf ("Missing argument for option -%c\n",
5900078             optopt);
5900079         usage ();
5900080         return (2);
5900081         break;
5900082     default:
5900083         printf
5900084             ("Getopt problem: unknown option %c\n", opt);
5900085         return (3);
5900086     }
5900087 }
5900088 //
5900089 dir = argv + optind;
5900090 //
5900091 // Check if the mode is to be set to a default
5900092 // value.
5900093 //
5900094 if (!option_m)
5900095     {
5900096         //
5900097         // Default mode.
5900098         //
5900099         sys (SYS_UAREA, &msg, (sizeof msg));
5900100         mode = 0777 & ~msg.umask;
5900101     }
5900102 //
5900103 // Directory creation.
5900104 //
5900105 for (d = 0; dir[d] != NULL; d++)
5900106     {
5900107         if (option_p)
5900108             {
```

```
5900109         status = mkdir_parents (dir[d], mode);
5900110         if (status != 0)
5900111             {
5900112                 perror (dir[d]);
5900113                 return (3);
5900114             }
5900115     }
5900116     else
5900117     {
5900118         status = mkdir (dir[d], mode);
5900119         if (status != 0)
5900120             {
5900121                 perror (dir[d]);
5900122                 return (4);
5900123             }
5900124     }
5900125 }
5900126 //
5900127 // All done.
5900128 //
5900129 return (0);
5900130 }
5900131
5900132 //-----
5900133 static int
5900134 mkdir_parents (const char *path, mode_t mode)
5900135 {
5900136     char path_copy[PATH_MAX];
5900137     char *path_parent;
5900138     struct stat fst;
5900139     int status;
5900140     //
5900141     // Check if the path is empty.
5900142     //
5900143     if (path == NULL || strlen (path) == 0)
5900144         {
5900145             //
```

```
5900146      // Recursion ends here.
5900147      //
5900148      return (0);
5900149  }
5900150  //
5900151  // Check if it does already exists.
5900152  //
5900153  status = stat (path, &fst);
5900154  if (status == 0 && fst.st_mode & S_IFDIR)
5900155  {
5900156      //
5900157      // The path exists and is a directory.
5900158      //
5900159      return (0);
5900160  }
5900161  else if (status == 0 && !(fst.st_mode & S_IFDIR))
5900162  {
5900163      //
5900164      // The path exists but is not a directory.
5900165      //
5900166      errno = ENOTDIR; // Not a directory.
5900167      return (-1);
5900168  }
5900169  //
5900170  // Get the directory path.
5900171  //
5900172  strncpy (path_copy, path, PATH_MAX);
5900173  path_parent = dirname (path_copy);
5900174  //
5900175  // If it is '.', or '/', the recursion is
5900176  // terminated.
5900177  //
5900178  if (strncmp (path_parent, ".", PATH_MAX) == 0 ||
5900179      strncmp (path_parent, "/", PATH_MAX) == 0)
5900180  {
5900181      return (0);
5900182  }
```

```
5900183 //
5900184 // Otherwise, continue the recursion.
5900185 //
5900186 status = mkdir_parents (path_parent, mode);
5900187 if (status != 0)
5900188     {
5900189         return (-1);
5900190     }
5900191 //
5900192 // Previous directories are there: create the
5900193 // current one.
5900194 //
5900195 status = mkdir (path, mode);
5900196 if (status)
5900197     {
5900198         perror (path);
5900199         return (-1);
5900200     }
5900201
5900202 return (0);
5900203 }
5900204
5900205 //-----
5900206 static void
5900207 usage (void)
5900208 {
5900209     fprintf
5900210         (stderr, "Usage: mkdir [-p] [-m OCTAL_MODE] DIR...\n");
5900211 }
```

96.1.26 applic/mmcheck.c

Si veda la sezione [86.18](#).

```
5910001 #include <sys/os32.h>
5910002 #include <kernel/memory.h>
5910003 #include <kernel/proc.h>
```

```
5910004 #include <unistd.h>
5910005 #include <stdio.h>
5910006 #include <fcntl.h>
5910007 #include <unistd.h>
5910008 #include <stdlib.h>
5910009 //-----
5910010 uint32_t mb_table[MEM_MAX_BLOCKS / 32]; // Memory
5910011 // blocks map.
5910012 unsigned int mb_max = MEM_MAX_BLOCKS; // Memory
5910013 // blocks max.
5910014 proc_t process;
5910015 //-----
5910016 static int mb_block_set0 (int block);
5910017 static void mb_check (pid_t pid, addr_t address,
5910018                      size_t size);
5910019 static void mb_residual (void);
5910020 //-----
5910021 int
5910022 main (int argc, char *argv[], char *envp[])
5910023 {
5910024     int i;
5910025     int fd;
5910026     ssize_t size_read;
5910027     char *buffer;
5910028     pid_t pid;
5910029     proc_t *ps;
5910030     //
5910031     // Get memory map.
5910032     //
5910033     fd = open ("/dev/kmem_map", O_RDONLY);
5910034     if (fd < 0)
5910035     {
5910036         printf ("%s] Cannot open \"/dev/kmem_map\" ",
5910037                argv[0]);
5910038         perror (NULL);
5910039         return (0);
5910040     }
```



```
5910041 //
5910042 buffer = (char *) mb_table;
5910043 lseek (fd, (off_t) 0, SEEK_SET);
5910044 for (i = 0; i < (MEM_MAX_BLOCKS / 8); i += size_read)
5910045 {
5910046     size_read = read (fd, &buffer[i], BUFSIZ);
5910047     if (size_read < 0)
5910048     {
5910049         printf
5910050             ("[%s] Cannot read "
5910051              "\"/dev/kmem_map\" %i %i ",
5910052              argv[0], size_read, sizeof (mb_table));
5910053         perror (NULL);
5910054         return (0);
5910055     }
5910056 }
5910057 //
5910058 close (fd);
5910059 //
5910060 // Scan processes
5910061 //
5910062 buffer = (char *) &process;
5910063 //
5910064 fd = open ("/dev/kmem_ps", O_RDONLY);
5910065 if (fd < 0)
5910066 {
5910067     printf ("[%s] Cannot open \"/dev/kmem_ps\" ",
5910068            argv[0]);
5910069     perror (NULL);
5910070     exit (0);
5910071 }
5910072 //
5910073 // Scan processes.
5910074 //
5910075 for (pid = 0; pid < PROCESS_MAX; pid++)
5910076 {
5910077     lseek (fd, (off_t) pid, SEEK_SET);
```

```
5910078     size_read = read (fd, buffer, sizeof (proc_t));
5910079     if (size_read < sizeof (proc_t))
5910080     {
5910081         printf
5910082             ("[%s] Cannot read "
5910083              "\"/dev/kmem_ps\" pid %i ", argv[0], pid);
5910084         perror (NULL);
5910085         continue;
5910086     }
5910087     ps = (proc_t *) buffer;
5910088     if (ps->status > 0)
5910089     {
5910090         //
5910091         //
5910092         //
5910093         if (ps->domain_data == 0)
5910094         {
5910095             mb_check (pid, ps->address_text,
5910096                      ps->domain_text + ps->extra_data);
5910097         }
5910098         else
5910099         {
5910100             mb_check (pid, ps->address_text,
5910101                      ps->domain_text);
5910102             mb_check (pid, ps->address_data,
5910103                      ps->domain_data + ps->extra_data);
5910104         }
5910105     }
5910106 }
5910107 close (fd);
5910108 //
5910109 // Check residual allocation, if any.
5910110 //
5910111 mb_residual ();
5910112 //
5910113 return (0);
5910114 }
```

```
5910115
5910116 //-----
5910117 static void
5910118 mb_check (pid_t pid, addr_t address, size_t size)
5910119 {
5910120     unsigned int bstart;
5910121     unsigned int bsize;
5910122     unsigned int bend;
5910123     unsigned int i;
5910124     addr_t block_address;
5910125     //
5910126     // k_printf ("releasing 0x%x, size 0x%x\n", (int)
5910127     // address,
5910128     // (int) size);
5910129     //
5910130     if (size == 0)
5910131     {
5910132         //
5910133         // Zero means nothing.
5910134         //
5910135         return;
5910136     }
5910137     //
5910138     if (size % MEM_BLOCK_SIZE)
5910139     {
5910140         bsize = size / MEM_BLOCK_SIZE + 1;
5910141     }
5910142     else
5910143     {
5910144         bsize = size / MEM_BLOCK_SIZE;
5910145     }
5910146     //
5910147     bstart = address / MEM_BLOCK_SIZE;
5910148     bend = bstart + bsize;
5910149     //
5910150     //
5910151     //
```



```
5910189 }
5910190
5910191 //-----
5910192 static void
5910193 mb_residual (void)
5910194 {
5910195     unsigned int block;
5910196     unsigned int blocks = MEM_MAX_BLOCKS;
5910197     int i;
5910198     int j;
5910199     uint32_t mask;
5910200     unsigned int start = 0;
5910201     unsigned int stop = 0;
5910202     unsigned int status = 0;
5910203     //
5910204     // Show residual allocated memory.
5910205     //
5910206     for (block = 0; block < blocks; block++)
5910207     {
5910208         i = block / 32;
5910209         j = block % 32;
5910210         mask = 0x80000000 >> j;
5910211         if (mb_table[i] & mask)
5910212         {
5910213             //
5910214             // Allocated block
5910215             //
5910216             if (status == 0)
5910217             {
5910218                 status = 1;
5910219                 start = block;
5910220             }
5910221         }
5910222         else
5910223         {
5910224             //
5910225             // Not allocated block.
```

```
5910226         //
5910227         if (status == 1)
5910228             {
5910229                 status = 0;
5910230                 stop = block;
5910231             }
5910232     }
5910233     //
5910234     //
5910235     //
5910236     if (stop > 0)
5910237     {
5910238         printf ("residual allocation: %x-%x  ",
5910239                start, stop);
5910240         start = 0;
5910241         stop = 0;
5910242     }
5910243 }
5910244 printf ("\n");
5910245 }
```

96.1.27 applic/more.c



Si veda la sezione [86.19](#).

```
5920001 #include <unistd.h>
5920002 #include <errno.h>
5920003 //-----
5920004 #define MAX_LINES    20
5920005 #define MAX_COLUMNS  80
5920006 //-----
5920007 static void usage (void);
5920008 //-----
5920009 int
5920010 main (int argc, char *argv[], char *envp[])
5920011 {
5920012     FILE *fp;
```

```
5920013 char *name;
5920014 int c;
5920015 int line = 1; // Line internal counter.
5920016 int column = 1; // Column internal counter.
5920017 int a; // Index inside arguments.
5920018 int loop;
5920019 //
5920020 // There must be at least an argument, plus the
5920021 // program name.
5920022 //
5920023 if (argc < 2)
5920024 {
5920025     usage ();
5920026     return (1);
5920027 }
5920028 //
5920029 // No options are allowed.
5920030 //
5920031 for (a = 1; a < argc; a++)
5920032 {
5920033     //
5920034     // Get next name from arguments.
5920035     //
5920036     name = argv[a];
5920037     //
5920038     // Try to open the file, read only.
5920039     //
5920040     fp = fopen (name, "r");
5920041     //
5920042     if (fp == NULL)
5920043     {
5920044         //
5920045         // Error opening file.
5920046         //
5920047         perror (name);
5920048         return (1);
5920049     }
```

```
5920050 //
5920051 // Print the file name to be displayed.
5920052 //
5920053 printf ("== %s ==\n", name);
5920054 line++;
5920055 //
5920056 // The following loop continues while the file
5920057 // gives characters, or when a command to change
5920058 // file or to quit is given.
5920059 //
5920060 for (loop = 1; loop;)
5920061 {
5920062 //
5920063 // Read a single character.
5920064 //
5920065 c = getc (fp);
5920066 //
5920067 if (c == EOF)
5920068 {
5920069     loop = 0;
5920070     break;
5920071 }
5920072 //
5920073 // If the character read is a special one,
5920074 // the line/column calculation is modified,
5920075 // so that it is known when to stop
5920076 // scrolling.
5920077 //
5920078 switch (c)
5920079 {
5920080     case '\r':
5920081 //
5920082 // Displaying this character, the cursor
5920083 // should go
5920084 // back to the first column. So the
5920085 // column counter
5920086 // is reset.
```



```
5920087         //
5920088         column = 1;
5920089         break;
5920090     case '\n':
5920091         //
5920092         // Displaying this character, the cursor
5920093         // should go
5920094         // back to the next line, at the first
5920095         // column.
5920096         // So the column counter is reset and
5920097         // the line
5920098         // counter is incremented.
5920099         //
5920100         line++;
5920101         column = 1;
5920102         break;
5920103     case '\b':
5920104         //
5920105         // Displaying this character, the cursor
5920106         // should go
5920107         // back one position, unless it is
5920108         // already at the
5920109         // beginning.
5920110         //
5920111         if (column > 1)
5920112             {
5920113                 column--;
5920114             }
5920115         break;
5920116     default:
5920117         //
5920118         // Any other character must increase the
5920119         // column
5920120         // counter.
5920121         //
5920122         column++;
5920123     }
```

```
5920124 //
5920125 // Display the character, even if it is a
5920126 // special one:
5920127 // it is responsibility of the screen device
5920128 // management
5920129 // to do something good with special
5920130 // characters.
5920131 //
5920132 putchar (c);
5920133 //
5920134 // If the column counter is gone beyond the
5920135 // screen columns,
5920136 // then adjust the column counter and
5920137 // increment the line
5920138 // counter.
5920139 //
5920140 if (column > MAX_COLUMNS)
5920141     {
5920142         column -= MAX_COLUMNS;
5920143         line++;
5920144     }
5920145 //
5920146 // Check if there is space for scrolling.
5920147 //
5920148 if (line < MAX_LINES)
5920149     {
5920150         continue;
5920151     }
5920152 //
5920153 // Here, displayed lines are MAX_LINES.
5920154 //
5920155 if (column > 1)
5920156     {
5920157         //
5920158         // Something was printed at the current
5920159         // line: must
5920160         // do a new line.
```

```
5920161         //
5920162         putchar ('\n');
5920163     }
5920164     //
5920165     // Show the more prompt.
5920166     //
5920167     printf ("--More--");
5920168     fflush (stdout);
5920169     //
5920170     // Read a character from standard input.
5920171     //
5920172     c = getchar ();
5920173     //
5920174     // Consider commands 'n' and 'q', but any
5920175     // other character
5920176     // can be introduced, to let show the next
5920177     // page.
5920178     //
5920179     switch (c)
5920180     {
5920181     case 'N':
5920182     case 'n':
5920183         //
5920184         // Go to the next file, if any.
5920185         //
5920186         fclose (fp);
5920187         fp = NULL;
5920188         loop = 0;
5920189         break;
5920190     case 'Q':
5920191     case 'q':
5920192         // //
5920193         // // Quit. But must erase the
5920194         // '--More--' prompt.
5920195         // //
5920196         // printf ("\b \b\b \b\b \b\b \b\b \b\b \b");
5920197         // printf ("\b \b\b \b\b \b\b \b");
```


96.1.28 applic/mount.c



Si veda la sezione [92.7](#).

```
5930001 #include <unistd.h>
5930002 #include <stdlib.h>
5930003 #include <sys/stat.h>
5930004 #include <sys/types.h>
5930005 #include <fcntl.h>
5930006 #include <errno.h>
5930007 #include <signal.h>
5930008 #include <stdio.h>
5930009 #include <sys/wait.h>
5930010 #include <stdio.h>
5930011 #include <string.h>
5930012 #include <limits.h>
5930013 #include <sys/os32.h>
5930014 //-----
5930015 static void usage (void);
5930016 //-----
5930017 int
5930018 main (int argc, char *argv[], char *envp[])
5930019 {
5930020     int options;
5930021     int status;
5930022     //
5930023     //
5930024     //
5930025     if (argc < 3 || argc > 4)
5930026     {
5930027         usage ();
5930028         return (1);
5930029     }
5930030     //
5930031     // Set options.
5930032     //
5930033     if (argc == 4)
5930034     {
```

```
5930035     if (strcmp (argv[3], "rw") == 0)
5930036     {
5930037         options = MOUNT_DEFAULT;
5930038     }
5930039     else if (strcmp (argv[3], "ro") == 0)
5930040     {
5930041         options = MOUNT_RO;
5930042     }
5930043     else
5930044     {
5930045         printf
5930046             ("Invalid mount option: "
5930047              "only \"ro\" or \"rw\" " "are allowed\n");
5930048         return (2);
5930049     }
5930050 }
5930051 else
5930052 {
5930053     options = MOUNT_DEFAULT;
5930054 }
5930055 //
5930056 // System call.
5930057 //
5930058 status = mount (argv[1], argv[2], options);
5930059 if (status != 0)
5930060 {
5930061     perror (NULL);
5930062     return (2);
5930063 }
5930064 //
5930065 return (0);
5930066 }
5930067
5930068 //-----
5930069 static void
5930070 usage (void)
5930071 {
```

```
5930072     fprintf (stderr, "Usage: mount DEVICE MOUNT_POINT "  
5930073             "[MOUNT_OPTIONS]\n");  
5930074 }
```

96.1.29 applic/nc.c

Si veda la sezione [86.20](#).

```
5940001 #include <sys/stat.h>  
5940002 #include <sys/types.h>  
5940003 #include <unistd.h>  
5940004 #include <stdlib.h>  
5940005 #include <fcntl.h>  
5940006 #include <errno.h>  
5940007 #include <signal.h>  
5940008 #include <stdio.h>  
5940009 #include <string.h>  
5940010 #include <limits.h>  
5940011 #include <libgen.h>  
5940012 #include <arpa/inet.h>  
5940013 #include <sys/socket.h>  
5940014 #include <stdint.h>  
5940015 #include <stdbool.h>  
5940016 #include <fcntl.h>  
5940017 //-----  
5940018 static void usage (void);  
5940019 char buffer[BUFSIZ];  
5940020 //-----  
5940021 int  
5940022 main (int argc, char *argv[], char *envp[])  
5940023 {  
5940024     bool option_l = 0;  
5940025     bool option_u = 0;  
5940026     int opt;  
5940027     //extern char *optarg;           // not used.  
5940028     extern int optind;  
5940029     extern int optopt;
```

```
5940030 //
5940031 int status;
5940032 int sfdn;
5940033 int sfdn2;
5940034 struct sockaddr_in sa_local;
5940035 struct sockaddr_in sa_remote;
5940036 socklen_t sa_remote_size = sizeof (struct sockaddr_in);
5940037 ssize_t read_size;
5940038 ssize_t sent_size;
5940039 ssize_t recv_size;
5940040 char *addr = NULL;
5940041 char *port = NULL;
5940042 bool can_rx = 1;
5940043 bool can_tx = 1;
5940044 //
5940045 // Check for options.
5940046 //
5940047 while ((opt = getopt (argc, argv, ":ul")) != -1)
5940048     {
5940049     switch (opt)
5940050     {
5940051     case 'l':
5940052         option_l = 1;
5940053         break;
5940054     case 'u':
5940055         option_u = 1;
5940056         break;
5940057     case '?':
5940058         fprintf (stderr, "Unknown option -%c.\n", optopt);
5940059         usage ();
5940060         return (1);
5940061         break;
5940062     case ':':
5940063         fprintf (stderr,
5940064                 "Missing argument for option -%c\n",
5940065                 optopt);
5940066         usage ();
```



```
5940067         return (1);
5940068         break;
5940069     default:
5940070         fprintf (stderr,
5940071                 "Getopt problem: "
5940072                 "unknown option %c\n", opt);
5940073         usage ();
5940074         return (1);
5940075     }
5940076 }
5940077 //
5940078 // Arguments.
5940079 //
5940080 if (optind == (argc - 2))
5940081 {
5940082     //
5940083     // There are exactly two arguments: destination
5940084     // address and port.
5940085     //
5940086     addr = argv[argc - 2];
5940087     port = argv[argc - 1];
5940088 }
5940089 else
5940090 {
5940091     //
5940092     // Arguments wrong!
5940093     //
5940094     usage ();
5940095     return (2);
5940096 }
5940097 //
5940098 // Set the local or the remote address.
5940099 //
5940100 if (option_l)
5940101 {
5940102     //
5940103     // Address and port are local.
```

```
5940104      //
5940105      sa_local.sin_family = AF_INET;
5940106      sa_local.sin_port = htons (atoi (port));
5940107      inet_pton (AF_INET, addr, &sa_local.sin_addr.s_addr);
5940108  }
5940109  else
5940110  {
5940111      //
5940112      // Address and port are remote.
5940113      //
5940114      sa_remote.sin_family = AF_INET;
5940115      sa_remote.sin_port = htons (atoi (port));
5940116      inet_pton (AF_INET, addr, &sa_remote.sin_addr.s_addr);
5940117  }
5940118  //
5940119  // Open the socket.
5940120  //
5940121  if (option_u)
5940122  {
5940123      sfdn = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
5940124  }
5940125  else
5940126  {
5940127      sfdn = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
5940128  }
5940129  if (sfdn < 0)
5940130  {
5940131      perror (NULL);
5940132      return (3);
5940133  }
5940134  //
5940135  // Set it listening or connect.
5940136  //
5940137  if (option_l)
5940138  {
5940139      //
5940140      // Bind the local 'sa' location.
```

```
5940141 //
5940142 status =
5940143     bind (sfdn, (struct sockaddr *) &sa_local,
5940144           sizeof (sa_local));
5940145 if (status < 0)
5940146     {
5940147     perror (NULL);
5940148     close (sfdn);
5940149     return (4);
5940150     }
5940151 //
5940152 // Listen (TCP) or wait the first packet (UDP).
5940153 //
5940154 if (option_u)
5940155     {
5940156     //
5940157     // Instead of listening, we use the function
5940158     // 'recvfrom()',
5940159     // to get the remote address and port.
5940160     //
5940161     recv_size =
5940162         recvfrom (sfdn, &buffer,
5940163                  (size_t) BUFSIZ - 1, 0,
5940164                  (struct sockaddr *) &sa_remote,
5940165                  &sa_remote_size);
5940166     if (recv_size < 0)
5940167         {
5940168         perror (NULL);
5940169         close (sfdn);
5940170         return (4);
5940171         }
5940172 //
5940173 // Now connect the remote destination
5940174 //
5940175 status =
5940176     connect (sfdn,
5940177             (struct sockaddr *) &sa_remote,
```

```
5940178         sizeof (sa_remote));
5940179     if (status < 0)
5940180     {
5940181         perror (NULL);
5940182         close (sfdn);
5940183         return (7);
5940184     }
5940185     //
5940186     // And show what was received as a first
5940187     // packet.
5940188     //
5940189     buffer[recv_size] = 0;
5940190     printf ("%s", buffer);
5940191 }
5940192 else
5940193 {
5940194     //
5940195     // TCP: listen.
5940196     //
5940197     status = listen (sfdn, 1);
5940198     if (status < 0)
5940199     {
5940200         perror (NULL);
5940201         close (sfdn);
5940202         return (5);
5940203     }
5940204     //
5940205     // Accept.
5940206     //
5940207     sfdn2 =
5940208         accept (sfdn,
5940209             (struct sockaddr *) &sa_remote,
5940210             &sa_remote_size);
5940211     if (sfdn2 < 0)
5940212     {
5940213         perror (NULL);
5940214         close (sfdn);
```

```
5940215         return (6);
5940216     }
5940217     //
5940218     // Close listening socket.
5940219     //
5940220     close (sfdn);
5940221     //
5940222     // Variable 'sfdn' will be the new socket.
5940223     //
5940224     sfdn = sfdn2;
5940225 }
5940226 }
5940227 else
5940228 {
5940229     //
5940230     // Connect the remote destination.
5940231     //
5940232     status =
5940233         connect (sfdn, (struct sockaddr *) &sa_remote,
5940234                 sizeof (sa_remote));
5940235     if (status < 0)
5940236     {
5940237         perror (NULL);
5940238         close (sfdn);
5940239         return (7);
5940240     }
5940241 }
5940242 //
5940243 // Define the standard input non blocking.
5940244 //
5940245 status = fcntl (STDIN_FILENO, F_SETFL, O_NONBLOCK);
5940246 if (status < 0)
5940247 {
5940248     perror (NULL);
5940249     return (8);
5940250 }
5940251 //
```

```
5940252 // Define the socket non blocking.
5940253 //
5940254 status = fcntl (sfdn, F_SETFL, O_NONBLOCK);
5940255 if (status < 0)
5940256 {
5940257     perror (NULL);
5940258     return (9);
5940259 }
5940260 //
5940261 // Will read from the remote and show to the screen.
5940262 //
5940263 while (can_rx || can_tx)
5940264 {
5940265     if (can_rx)
5940266     {
5940267         recv_size =
5940268             recv (sfdn, &buffer, (size_t) BUFSIZ - 1, 0);
5940269 //         recv_size = read (sfdn, &buffer,
5940270 //             (size_t) BUFSIZ-1);
5940271         if (recv_size < 0)
5940272         {
5940273             if (errno == EAGAIN || errno == EWOULDBLOCK)
5940274             {
5940275                 ;
5940276             }
5940277             else
5940278             {
5940279                 perror (NULL);
5940280                 close (sfdn);
5940281                 return (10);
5940282             }
5940283         }
5940284         else if (recv_size == 0)
5940285         {
5940286             //
5940287             // End of stream.
5940288             //
```

```
5940289         can_rx = 0;
5940290         printf ("--end of receive stream--\n");
5940291     }
5940292     else
5940293     {
5940294         buffer[recv_size] = 0;
5940295         printf ("%s", buffer);
5940296     }
5940297 }
5940298 if (can_tx)
5940299 {
5940300     read_size = read (STDIN_FILENO, buffer, BUFSIZ);
5940301     if (read_size < 0)
5940302     {
5940303         if (errno == EAGAIN || errno == EWOULDBLOCK)
5940304         {
5940305             ;
5940306         }
5940307         else
5940308         {
5940309             perror (NULL);
5940310             close (sfdn);
5940311             return (11);
5940312         }
5940313     }
5940314     else if (read_size == 0)
5940315     {
5940316         //
5940317         // End of input.
5940318         //
5940319         printf ("--closing send stream--\n");
5940320         can_tx = 0;
5940321     }
5940322     else
5940323     {
5940324         //
5940325         // Send it.
```

```
5940326         //
5940327         sent_size =
5940328             send (sfdn, &buffer, (size_t) read_size, 0);
5940329         if (sent_size < 0)
5940330             {
5940331                 if (errno == EAGAIN
5940332                     || errno == EWOULDBLOCK)
5940333                     {
5940334                         ;
5940335                     }
5940336                 else
5940337                     {
5940338                         perror (NULL);
5940339                         close (sfdn);
5940340                         return (12);
5940341                     }
5940342             }
5940343         }
5940344     }
5940345 }
5940346 //
5940347 // All done.
5940348 //
5940349 close (sfdn);
5940350 return (0);
5940351 }
5940352
5940353 //-----
5940354 static void
5940355 usage (void)
5940356 {
5940357     fprintf
5940358         (stderr,
5940359          "os32 netcat usage:\n"
5940360          "\n"
5940361          "nc [-u][-l] ADDRESS PORT\n"
5940362          "\n"
```



```

5940363     "-u      Use UDP protocol instead of TCP.\n"
5940364     "-l      Listen for incoming connection \n"
5940365     "      requests.\n"
5940366     "ADDRESS IPv4 numeric address; if option -l is\n"
5940367     "      used, this\n"
5940368     "      is the local address, otherwise it is\n"
5940369     "      the remote address.\n"
5940370     "PORT   TCP or UDP port; if option -l is used,\n"
5940371     "      this is local address, otherwise it is\n"
5940372     "      the remote address.\n");
5940373 }

```

96.1.30 applic/ping.c

Si veda la sezione [92.8](#).



```

5950001 #include <stdio.h>
5950002 #include <sys/types.h>
5950003 #include <arpa/inet.h>
5950004 #include <sys/socket.h>
5950005 #include <netinet/icmp.h>
5950006 #include <unistd.h>
5950007 #include <stdlib.h>
5950008 #include <stdint.h>
5950009 #include <errno.h>
5950010 //-----
5950011 struct ip_pkt
5950012 {
5950013     struct iphdr ip;      // <netinet/ip.h>
5950014     struct icmphdr icmp; // <netinet/icmp.h>
5950015     char data[60];
5950016 } __attribute__((packed));
5950017
5950018 struct icmp_pkt
5950019 {
5950020     struct icmphdr icmp; // <netinet/icmp.h>
5950021     char data[60];

```

```
5950022 } __attribute__ ((packed));
5950023 //-----
5950024 static uint16_t ip_chk (uint16_t * data, size_t size);
5950025 static void usage (void);
5950026 //-----
5950027 int
5950028 main (int argc, char *argv[], char *envp[])
5950029 {
5950030     int sfdn;
5950031     struct sockaddr_in sa;
5950032     ssize_t sent;
5950033     ssize_t received;
5950034     int status;
5950035     char *destination;
5950036     uint16_t checksum;
5950037     struct icmp_pkt icmp_pkt_send;
5950038     struct ip_pkt ip_pkt_receive;
5950039     //int                retry = 3;                // Max send retry.
5950040     clock_t clock_ping;
5950041     clock_t clock_pong;
5950042     clock_t clock_time;
5950043     //
5950044     // No options are known, but an argument must be
5950045     // given.
5950046     //
5950047     if (argc < 2)
5950048     {
5950049         usage ();
5950050         return (1);
5950051     }
5950052     destination = argv[1];
5950053     //
5950054     // Define the destination 'sa'
5950055     //
5950056     sa.sin_family = AF_INET;
5950057     sa.sin_port = 0;
5950058     inet_pton (AF_INET, destination, &sa.sin_addr.s_addr);
```

```
5950059 //
5950060 // Put some data inside the packet header.
5950061 //
5950062 icmp_pkt_send.icmp.un.echo.id = (uint16_t) rand ();
5950063 icmp_pkt_send.icmp.un.echo.sequence = 0;
5950064 icmp_pkt_send.icmp.type = 8; // Echo request.
5950065 icmp_pkt_send.icmp.code = 0;
5950066 icmp_pkt_send.icmp.checksum = 0;
5950067 //
5950068 // Calculate the ICMP checksum.
5950069 //
5950070 checksum = ~(ip_chk ((void *) &icmp_pkt_send,
5950071                      sizeof (struct icmp_pkt)));
5950072 icmp_pkt_send.icmp.checksum = htons (checksum);
5950073 //
5950074 // Open the socket.
5950075 //
5950076 sfdn = socket (AF_INET, SOCK_RAW, IPPROTO_ICMP);
5950077 if (sfdn < 0)
5950078     {
5950079         perror (NULL);
5950080         return (0);
5950081     }
5950082 //
5950083 // Connect the 'sa' destination
5950084 //
5950085 status =
5950086     connect (sfdn, (struct sockaddr *) &sa, sizeof (sa));
5950087 if (status < 0)
5950088     {
5950089         perror (NULL);
5950090         close (sfdn);
5950091         return (0);
5950092     }
5950093 //
5950094 // Send one single packet: please notice that we
5950095 // send an ICMP,
```

```
5950096 // but we receive a full IP.
5950097 //
5950098 clock_ping = clock ();
5950099 sent = send (sfdn, &icmp_pkt_send,
5950100             sizeof (struct icmp_pkt), 0);
5950101 if (sent < 0)
5950102     {
5950103     perror (NULL);
5950104     return (1);
5950105     }
5950106 //
5950107 // Packet sent.
5950108 //
5950109 printf ("ping: ");
5950110 //
5950111 // Now receive all ICMP raw packets, and select the
5950112 // one with the same
5950113 // identifier.
5950114 //
5950115 while (1)
5950116     {
5950117     received =
5950118         read (sfdn, &ip_pkt_receive,
5950119             sizeof (struct ip_pkt));
5950120     clock_pong = clock ();
5950121
5950122     if (ip_pkt_receive.icmp.un.echo.id
5950123         == icmp_pkt_send.icmp.un.echo.id)
5950124         {
5950125         clock_time = (clock_pong - clock_ping);
5950126         printf ("pong %llu.%03llu s\n",
5950127             clock_time / CLOCKS_PER_SEC,
5950128             (clock_time % CLOCKS_PER_SEC) *
5950129             1000 / CLOCKS_PER_SEC);
5950130         break;
5950131         }
5950132     }
```

```
5950133
5950134     close (sfdn);
5950135     return (0);
5950136 }
5950137
5950138 //-----
5950139 static uint16_t
5950140 ip_chk (uint16_t * data, size_t size)
5950141 {
5950142     int i;
5950143     uint32_t sum;
5950144     uint16_t carry;
5950145     uint16_t checksum;
5950146     uint16_t last;
5950147     uint8_t *octet = (uint8_t *) data;
5950148     //
5950149     // 2's complement sum.
5950150     //
5950151     for (i = 0, sum = 0; i < (size / 2); i++)
5950152     {
5950153         sum += ntohs (data[i]);
5950154     }
5950155     //
5950156     if (size % 2)
5950157     {
5950158         //
5950159         // The size is odd, and the last octet must be
5950160         // accounted too.
5950161         //
5950162         last = octet[size - 1];
5950163         last = last << 8;
5950164         //
5950165         sum += last;
5950166     }
5950167     //
5950168     // Extract the carries and make the checksum.
5950169     //
```

```
5950170     carry = sum >> 16;
5950171     checksum = sum & 0x0000FFFF;
5950172     checksum += carry;
5950173     //
5950174     // End of job.
5950175     //
5950176     return (checksum);
5950177 }
5950178
5950179 //-----
5950180 static void
5950181 usage (void)
5950182 {
5950183     fprintf (stderr, "Usage: ping IPv4\n");
5950184 }
```

96.1.31 applic/ps.c

<<

Si veda la sezione [86.21](#).

```
5960001 #include <sys/os32.h>
5960002 #include <kernel/proc.h>
5960003 #include <unistd.h>
5960004 #include <stdio.h>
5960005 #include <fcntl.h>
5960006 #include <unistd.h>
5960007 #include <stdlib.h>
5960008 //-----
5960009 static void usage (void);
5960010 //-----
5960011 int
5960012 main (int argc, char *argv[], char *envp[])
5960013 {
5960014     pid_t pid;
5960015     proc_t *ps;
5960016     int fd;
5960017     char stat;
```

```
5960018     ssize_t size_read;
5960019     char buffer[sizeof (proc_t)];
5960020     unsigned int min;
5960021     unsigned int sec;
5960022     size_t stack_size;
5960023     addr_t stack_bottom;
5960024     int stack_usage;
5960025     //
5960026     int opt;
5960027     // extern char *optarg;           // not used.
5960028     // extern int optind;
5960029     extern int optopt;
5960030     int option_u = 0;
5960031     int option_g = 0;
5960032     //
5960033     // Check for options.
5960034     //
5960035     while ((opt = getopt (argc, argv, ":ug")) != -1)
5960036     {
5960037         switch (opt)
5960038         {
5960039             case 'u':
5960040                 option_u = 1;
5960041                 break;
5960042             case 'g':
5960043                 option_g = 1;
5960044                 break;
5960045             case '?':
5960046                 fprintf (stderr, "Unknown option -%c.\n", optopt);
5960047                 usage ();
5960048                 return (1);
5960049                 break;
5960050             case ':':
5960051                 fprintf (stderr,
5960052                     "Missing argument for option -%c\n",
5960053                     optopt);
5960054                 usage ();
```

```
5960055         return (1);
5960056         break;
5960057     default:
5960058         fprintf (stderr,
5960059                 "Getopt problem: "
5960060                 "unknown option %c\n", opt);
5960061         return (1);
5960062     }
5960063 }
5960064 //
5960065 // Fix options '-u' or '-g'.
5960066 //
5960067 if (option_u)
5960068     option_g = 0;
5960069 if (!option_g)
5960070     option_u = 1;
5960071 //
5960072 // Open '/dev/kmem_ps', to get the process table.
5960073 //
5960074 fd = open ("/dev/kmem_ps", O_RDONLY);
5960075 if (fd < 0)
5960076     {
5960077         printf ("%s] Cannot open \"/dev/kmem_ps\" ",
5960078                 argv[0]);
5960079         perror (NULL);
5960080         exit (0);
5960081     }
5960082 //
5960083 // Print head.
5960084 //
5960085 if (option_u)
5960086     {
5960087         printf ("pp  p pg                "
5960088                 "T * 0x1000 D * 0x1000 stack        \n"
5960089                 "id id rp  tty  uid  euid  suid  usage  s  "
5960090                 "addr  size addr  size usage        name\n");
5960091     }
```



```
5960092     else
5960093     {
5960094         printf ("pp p pg                                "
5960095                "T * 0x1000 D * 0x1000 stack          \n"
5960096                "id id rp tty gid egid sgid usage s "
5960097                "addr size addr size usage name\n");
5960098     }
5960099     //
5960100     // Scan processes and then print body.
5960101     //
5960102     for (pid = 0; pid < PROCESS_MAX; pid++)
5960103     {
5960104         lseek (fd, (off_t) pid, SEEK_SET);
5960105         size_read = read (fd, buffer, sizeof (proc_t));
5960106         if (size_read < sizeof (proc_t))
5960107         {
5960108             printf
5960109                 ("[%s] Cannot read "
5960110                  "\"/dev/kmem_ps\" pid %i ", argv[0], pid);
5960111             perror (NULL);
5960112             continue;
5960113         }
5960114         ps = (proc_t *) buffer;
5960115         if (ps->status > 0)
5960116         {
5960117             ps->name[PATH_MAX - 1] = 0;    // Terminated
5960118             // string.
5960119             //
5960120             // Check the current stack size.
5960121             //
5960122             if (ps->domain_data == 0)
5960123             {
5960124                 stack_bottom = ps->domain_text;
5960125             }
5960126             else
5960127             {
5960128                 stack_bottom = ps->domain_data;
```

```
5960129     }
5960130     //
5960131     stack_size = stack_bottom - ps->sp;
5960132     //
5960133     stack_usage = 100 * stack_size / ps->domain_stack;
5960134     //
5960135     switch (ps->status)
5960136     {
5960137     case PROC_EMPTY:
5960138         stat = '-';
5960139         break;
5960140     case PROC_CREATED:
5960141         stat = 'c';
5960142         break;
5960143     case PROC_READY:
5960144         stat = 'r';
5960145         break;
5960146     case PROC_RUNNING:
5960147         stat = 'R';
5960148         break;
5960149     case PROC_SLEEPING:
5960150         stat = 's';
5960151         break;
5960152     case PROC_ZOMBIE:
5960153         stat = 'z';
5960154         break;
5960155     default:
5960156         stat = '?';
5960157         break;
5960158     }
5960159     //
5960160     min = ((ps->usage / CLOCKS_PER_SEC) / 60);
5960161     sec = ((ps->usage / CLOCKS_PER_SEC) % 60);
5960162     //
5960163     // Print the line.
5960164     //
5960165     //
```

```
5960166 // Addresses and sizes are multiple of 4096
5960167 // (0x1000);
5960168 // for the stack pointer is shown only the
5960169 // last five
5960170 // hexadecimal digits.
5960171 //
5960172 if (ps->domain_data > 0)
5960173 {
5960174     if (option_u)
5960175     {
5960176         printf
5960177             ("%2i %2i %2i %04x %4i %4i %4i "
5960178              "%02i.%02i %c %05x %04x %05x "
5960179              "%04x % 3i%% %15s\n",
5960180              (unsigned int) ps->ppid,
5960181              (unsigned int) pid,
5960182              (unsigned int) ps->pgrp,
5960183              (unsigned int) ps->device_tty,
5960184              (unsigned int) ps->uid,
5960185              (unsigned int) ps->euid,
5960186              (unsigned int) ps->suid, min, sec,
5960187              stat,
5960188              (unsigned int) ps->address_text /
5960189              MEM_BLOCK_SIZE,
5960190              (unsigned int) ps->domain_text /
5960191              MEM_BLOCK_SIZE,
5960192              (unsigned int) ps->address_data /
5960193              MEM_BLOCK_SIZE,
5960194              (unsigned int) (ps->domain_data +
5960195                           ps->extra_data) /
5960196              MEM_BLOCK_SIZE,
5960197              (unsigned int) stack_usage, ps->name);
5960198     }
5960199     else
5960200     {
5960201         printf
5960202             ("%2i %2i %2i %04x %4i %4i "
```

```
5960203         "%4i %02i.%02i %c %05x %04x "
5960204         "%05x %04x % 3i%% %15s\n",
5960205         (unsigned int) ps->ppid,
5960206         (unsigned int) pid,
5960207         (unsigned int) ps->pgrp,
5960208         (unsigned int) ps->device_tty,
5960209         (unsigned int) ps->gid,
5960210         (unsigned int) ps->egid,
5960211         (unsigned int) ps->sgid, min, sec,
5960212         stat,
5960213         (unsigned int) ps->address_text /
5960214         MEM_BLOCK_SIZE,
5960215         (unsigned int) ps->domain_text /
5960216         MEM_BLOCK_SIZE,
5960217         (unsigned int) ps->address_data /
5960218         MEM_BLOCK_SIZE,
5960219         (unsigned int) (ps->domain_data +
5960220                         ps->extra_data) /
5960221         MEM_BLOCK_SIZE,
5960222         (unsigned int) stack_usage, ps->name);
5960223     }
5960224 }
5960225 else
5960226 {
5960227     if (option_u)
5960228     {
5960229         printf
5960230         ("%2i %2i %2i %04x %4i %4i %4i "
5960231         "%02i.%02i %c %05x %04x %05x "
5960232         "%04x % 3i%% %15s\n",
5960233         (unsigned int) ps->ppid,
5960234         (unsigned int) pid,
5960235         (unsigned int) ps->pgrp,
5960236         (unsigned int) ps->device_tty,
5960237         (unsigned int) ps->uid,
5960238         (unsigned int) ps->euid,
5960239         (unsigned int) ps->suid, min, sec,
```

```
5960240         stat,
5960241         (unsigned int) ps->address_text /
5960242 MEM_BLOCK_SIZE,
5960243         (unsigned int) (ps->domain_text +
5960244                         ps->extra_data) /
5960245 MEM_BLOCK_SIZE,
5960246         (unsigned int) ps->address_data /
5960247 MEM_BLOCK_SIZE,
5960248         (unsigned int) ps->domain_data /
5960249 MEM_BLOCK_SIZE,
5960250         (unsigned int) stack_usage, ps->name);
5960251     }
5960252     else
5960253     {
5960254         printf
5960255         ("%2i %2i %2i %04x %4i %4i %4i "
5960256         "%02i.%02i %c %05x %04x %05x "
5960257         "%04x % 3i%% %15s\n",
5960258         (unsigned int) ps->ppid,
5960259         (unsigned int) pid,
5960260         (unsigned int) ps->pgrp,
5960261         (unsigned int) ps->device_tty,
5960262         (unsigned int) ps->gid,
5960263         (unsigned int) ps->egid,
5960264         (unsigned int) ps->sgid, min, sec,
5960265         stat,
5960266         (unsigned int) ps->address_text /
5960267 MEM_BLOCK_SIZE,
5960268         (unsigned int) (ps->domain_text +
5960269                         ps->extra_data) /
5960270 MEM_BLOCK_SIZE,
5960271         (unsigned int) ps->address_data /
5960272 MEM_BLOCK_SIZE,
5960273         (unsigned int) ps->domain_data /
5960274 MEM_BLOCK_SIZE,
5960275         (unsigned int) stack_usage, ps->name);
5960276     }
```

```
5960277     }
5960278     }
5960279     }
5960280     close (fd);
5960281     return (0);
5960282 }
5960283
5960284 //-----
5960285 static void
5960286 usage (void)
5960287 {
5960288     fprintf (stderr, "Usage: ps [-u|g]\n");
5960289 }
```

96.1.32 applic/rm.c



Si veda la sezione [86.22](#).

```
5970001 #include <fcntl.h>
5970002 #include <sys/stat.h>
5970003 #include <stddef.h>
5970004 #include <unistd.h>
5970005 #include <errno.h>
5970006 //-----
5970007 static void usage (void);
5970008 //-----
5970009 int
5970010 main (int argc, char *argv[], char *envp[])
5970011 {
5970012     int a;           // Argument index.
5970013     int status;
5970014     struct stat file_status;
5970015     //
5970016     // No options are known, but at least an argument
5970017     // must be given.
5970018     //
5970019     if (argc < 2)
```

```
5970020     {
5970021         usage ();
5970022         return (1);
5970023     }
5970024     //
5970025     // Scan arguments.
5970026     //
5970027     for (a = 1; a < argc; a++)
5970028     {
5970029         //
5970030         // Verify if the file exists.
5970031         //
5970032         if (stat (argv[a], &file_status) != 0)
5970033         {
5970034             fprintf (stderr,
5970035                     "File \"%s\" does not exist!\n",
5970036                     argv[a]);
5970037             continue;
5970038         }
5970039         //
5970040         // File exists: check the file type.
5970041         //
5970042         if (S_ISDIR (file_status.st_mode))
5970043         {
5970044             fprintf (stderr,
5970045                     "Cannot remove directory \"%s\"!\n",
5970046                     argv[a]);
5970047             continue;
5970048         }
5970049         //
5970050         // Can remove it.
5970051         //
5970052         status = unlink (argv[a]);
5970053         if (status != 0)
5970054         {
5970055             perror (NULL);
5970056             return (2);
```

```
5970057     }
5970058     }
5970059     return (0);
5970060 }
5970061
5970062 //-----
5970063 static void
5970064 usage (void)
5970065 {
5970066     fprintf (stderr, "Usage: rm FILE...\n");
5970067 }
```

96.1.33 applic/rmdir.c

<<

Si veda la sezione [86.23](#).

```
5980001 #include <fcntl.h>
5980002 #include <sys/stat.h>
5980003 #include <stddef.h>
5980004 #include <unistd.h>
5980005 #include <errno.h>
5980006 //-----
5980007 static void usage (void);
5980008 //-----
5980009 int
5980010 main (int argc, char *argv[], char *envp[])
5980011 {
5980012     int a;           // Argument index.
5980013     int status;
5980014     struct stat file_status;
5980015     //
5980016     // No options are known, but at least an argument
5980017     // must be given.
5980018     //
5980019     if (argc < 2)
5980020     {
5980021         usage ();
```



```
5980022     return (1);
5980023 }
5980024 //
5980025 // Scan arguments.
5980026 //
5980027 for (a = 1; a < argc; a++)
5980028 {
5980029     //
5980030     // Verify if the file exists.
5980031     //
5980032     if (stat (argv[a], &file_status) != 0)
5980033     {
5980034         fprintf (stderr,
5980035                 "File \"%s\" does not exist!\n",
5980036                 argv[a]);
5980037         continue;
5980038     }
5980039     //
5980040     // File exists: check the file type.
5980041     //
5980042     if (!S_ISDIR (file_status.st_mode))
5980043     {
5980044         fprintf (stderr,
5980045                 "Cannot remove file \"%s\"!\n", argv[a]);
5980046         continue;
5980047     }
5980048     //
5980049     // Can try to remove it.
5980050     //
5980051     status = rmdir (argv[a]);
5980052     if (status != 0)
5980053     {
5980054         perror (NULL);
5980055         return (2);
5980056     }
5980057 }
5980058 return (0);
```

```
5980059 }
5980060
5980061 //-----
5980062 static void
5980063 usage (void)
5980064 {
5980065     fprintf (stderr, "Usage: rmdir DIR...\n");
5980066 }
```

96.1.34 applic/route.c



Si veda la sezione [92.9](#).

```
5990001 #include <sys/os32.h>
5990002 #include <kernel/net/route.h>
5990003 #include <kernel/net.h>
5990004 #include <unistd.h>
5990005 #include <stdio.h>
5990006 #include <fcntl.h>
5990007 #include <unistd.h>
5990008 #include <stdlib.h>
5990009 //-----
5990010 int
5990011 main (int argc, char *argv[], char *envp[])
5990012 {
5990013     int fd;
5990014     ssize_t size_read;
5990015     char buffer[sizeof (route_t)];
5990016     int n;
5990017     route_t *route_table_item;
5990018     char string[80];
5990019
5990020     //
5990021     // All options are ignored, at the moment
5990022     //
5990023
5990024     //
```

```
5990025 // Open '/dev/kmem_route', to get the routing table.
5990026 //
5990027 fd = open ("/dev/kmem_route", O_RDONLY);
5990028 if (fd < 0)
5990029 {
5990030     printf ("%s] Cannot open \"/dev/kmem_route\" ",
5990031             argv[0]);
5990032     perror (NULL);
5990033     exit (0);
5990034 }
5990035 //
5990036 // Print header.
5990037 //
5990038 printf ("Destination/mask      "
5990039         "Router                " "Interface\n");
5990040 //
5990041 // Scan route table items and then print body.
5990042 //
5990043 for (n = 0; n < ROUTE_MAX_ROUTES; n++)
5990044 {
5990045     lseek (fd, (off_t) n, SEEK_SET);
5990046     size_read = read (fd, buffer, sizeof (route_t));
5990047     if (size_read < sizeof (route_t))
5990048     {
5990049         printf
5990050             ("%s] Cannot read "
5990051              "\"/dev/kmem_route\" item %i ", argv[0], n);
5990052         perror (NULL);
5990053         continue;
5990054     }
5990055     //
5990056     route_table_item = (route_t *) buffer;
5990057     //
5990058     if (route_table_item->network == 0xFFFFFFFF)
5990059     {
5990060         //
5990061         // Empty item.
```

```
5990062         //
5990063         continue;
5990064     }
5990065     //
5990066     sprintf (string, "%i.%i.%i.%i/%i"
5990067             "           ",
5990068             route_table_item->network >> 24 & 0x000000FF,
5990069             route_table_item->network >> 16 & 0x000000FF,
5990070             route_table_item->network >> 8 & 0x000000FF,
5990071             route_table_item->network >> 0 & 0x000000FF,
5990072             route_table_item->m);
5990073     string[19] = '\0';
5990074     printf ("%s", string);
5990075     //
5990076     if (route_table_item->router == 0)
5990077     {
5990078         printf ("           ");
5990079     }
5990080     else
5990081     {
5990082         sprintf (string, "%i.%i.%i.%i"
5990083                 "           ",
5990084                 route_table_item->router >> 24 &
5990085                 0x000000FF,
5990086                 route_table_item->router >> 16 &
5990087                 0x000000FF,
5990088                 route_table_item->router >> 8 &
5990089                 0x000000FF,
5990090                 route_table_item->router >> 0 &
5990091                 0x000000FF);
5990092         string[16] = '\0';
5990093         printf ("%s", string);
5990094     }
5990095     //
5990096     printf ("net%i\n", route_table_item->interface);
5990097 }
5990098 close (fd);
```

```
5990099     return (0);
5990100     }
```

96.1.35 applic/shell.c

Si veda la sezione [86.24](#).

```
6000001     #include <unistd.h>
6000002     #include <stdlib.h>
6000003     #include <sys/stat.h>
6000004     #include <sys/types.h>
6000005     #include <fcntl.h>
6000006     #include <errno.h>
6000007     #include <unistd.h>
6000008     #include <signal.h>
6000009     #include <stdio.h>
6000010     #include <sys/wait.h>
6000011     #include <stdio.h>
6000012     #include <string.h>
6000013     #include <limits.h>
6000014     #include <sys/os32.h>
6000015     //-----
6000016     #define PROMPT_SIZE      16
6000017     //-----
6000018     static void sh_cd (int argc, char *argv[]);
6000019     static void sh_pwd (int argc, char *argv[]);
6000020     static void sh_umask (int argc, char *argv[]);
6000021     //-----
6000022     int
6000023     main (int argc, char *argv[], char *envp[])
6000024     {
6000025         char buffer_cmd[ARG_MAX / 2];
6000026         char *argv_cmd[ARG_MAX / 16];
6000027         //char  prompt[PROMPT_SIZE];
6000028         uid_t uid;
6000029         int argc_cmd;
6000030         pid_t pid_cmd;
```

```
600031 pid_t pid_dead;
600032 int status;
600033 void *pstatus;
600034 int i;
600035 //
600036 //
600037 //
600038 uid = geteuid ();
600039 //
600040 // Load processes, reading the keyboard.
600041 //
600042 while (1)
600043 {
600044     if (uid == 0)
600045     {
600046         printf ("# ");
600047     }
600048     else
600049     {
600050         printf ("$ ");
600051     }
600052     //
600053     pstatus = fgets (buffer_cmd, (ARG_MAX / 2), stdin);
600054     if (pstatus == NULL)
600055     {
600056         if (errno)
600057         {
600058             perror (NULL);
600059             continue;
600060         }
600061         else
600062         {
600063             //
600064             // End of file, like ^D.
600065             //
600066             return (0);
600067         }
600067     }
```

```
600068     }
600069     //
600070     i = strlen (buffer_cmd);
600071     if (i > 0 && buffer_cmd[i - 1] == '\n')
600072     {
600073         buffer_cmd[i - 1] = '\0';
600074     }
600075     //
600076     // Clear 'argv_cmd[]';
600077     //
600078     for (argc_cmd = 0; argc_cmd < (ARG_MAX / 16);
600079         argc_cmd++)
600080     {
600081         argv_cmd[argc_cmd] = NULL;
600082     }
600083     //
600084     // Initialize the command scan.
600085     //
600086     argv_cmd[0] = strtok (buffer_cmd, " \t");
600087     //
600088     // Verify: if the input is not valid, loop
600089     // again.
600090     //
600091     if (argv_cmd[0] == NULL)
600092     {
600093         continue;
600094     }
600095     //
600096     // Find the arguments.
600097     //
600098     for (argc_cmd = 1;
600099         argc_cmd < ((ARG_MAX / 16) - 1)
600100         && argv_cmd[argc_cmd - 1] != NULL; argc_cmd++)
600101     {
600102         argv_cmd[argc_cmd] = strtok (NULL, " \t");
600103     }
600104     //
```

```
6000105 // If there are too many arguments, show a
6000106 // message and continue.
6000107 //
6000108 if (argv_cmd[argc_cmd - 1] != NULL)
6000109     {
6000110         errset (E2BIG); // Argument list too
6000111         // long.
6000112         perror (NULL);
6000113         continue;
6000114     }
6000115 //
6000116 // Correct the value for 'argc_cmd', because
6000117 // actually
6000118 // it counts also the NULL element.
6000119 //
6000120 argc_cmd--;
6000121 //
6000122 // Verify if it is an internal command.
6000123 //
6000124 if (strcmp (argv_cmd[0], "exit") == 0)
6000125     {
6000126         return (0);
6000127     }
6000128 else if (strcmp (argv_cmd[0], "cd") == 0)
6000129     {
6000130         sh_cd (argc_cmd, argv_cmd);
6000131         continue;
6000132     }
6000133 else if (strcmp (argv_cmd[0], "pwd") == 0)
6000134     {
6000135         sh_pwd (argc_cmd, argv_cmd);
6000136         continue;
6000137     }
6000138 else if (strcmp (argv_cmd[0], "umask") == 0)
6000139     {
6000140         sh_umask (argc_cmd, argv_cmd);
6000141         continue;
```



```
6000142     }
6000143     //
6000144     // It should be a program to run.
6000145     //
6000146     pid_cmd = fork ();
6000147     if (pid_cmd == -1)
6000148     {
6000149         printf ("%s: cannot run command", argv[0]);
6000150         perror (NULL);
6000151     }
6000152     else if (pid_cmd == 0)
6000153     {
6000154         execvp (argv_cmd[0], argv_cmd);
6000155         perror (NULL);
6000156         exit (0);
6000157     }
6000158     while (1)
6000159     {
6000160         pid_dead = wait (&status);
6000161         if (pid_dead == pid_cmd)
6000162         {
6000163             break;
6000164         }
6000165     }
6000166     printf ("pid %i terminated with status %i.\n",
6000167           (int) pid_dead, status);
6000168 }
6000169 }
6000170
6000171 //-----
6000172 static void
6000173 sh_cd (int argc, char *argv[])
6000174 {
6000175     int status;
6000176     //
6000177     if (argc != 2)
6000178     {
```

```
6000179     errset (EINVAL); // Invalid argument.
6000180     perror (NULL);
6000181     return;
6000182 }
6000183 //
6000184 status = chdir (argv[1]);
6000185 if (status != 0)
6000186 {
6000187     perror (NULL);
6000188 }
6000189 return;
6000190 }
6000191
6000192 //-----
6000193 static void
6000194 sh_pwd (int argc, char *argv[])
6000195 {
6000196     char path[PATH_MAX];
6000197     void *pstatus;
6000198     //
6000199     if (argc != 1)
6000200     {
6000201         errset (EINVAL); // Invalid argument.
6000202         perror (NULL);
6000203         return;
6000204     }
6000205     //
6000206     // Get the current directory.
6000207     //
6000208     pstatus = getcwd (path, (size_t) PATH_MAX);
6000209     if (pstatus == NULL)
6000210     {
6000211         perror (NULL);
6000212     }
6000213     else
6000214     {
6000215         printf ("%s\n", path);
```

```
6000216     }
6000217     return;
6000218 }
6000219
6000220 //-----
6000221 static void
6000222 sh_umask (int argc, char *argv[])
6000223 {
6000224     sysmsg_uarea_t msg;
6000225     char *m;      // Index inside the umask octal
6000226     // string.
6000227     int mask;
6000228     int digit;
6000229     //
6000230     if (argc > 2)
6000231     {
6000232         errset (EINVAL); // Invalid argument.
6000233         perror (NULL);
6000234         return;
6000235     }
6000236     //
6000237     // If no argument is available, the umask is shown,
6000238     // with a direct
6000239     // system call.
6000240     //
6000241     if (argc == 1)
6000242     {
6000243         sys (SYS_UAREA, &msg, (sizeof msg));
6000244         printf ("%04o\n", msg.umask);
6000245         return;
6000246     }
6000247     //
6000248     // Get the mask: must be the first argument.
6000249     //
6000250     for (mask = 0, m = argv[1]; *m != 0; m++)
6000251     {
6000252         digit = (*m - '0');
```

```
6000253     if (digit < 0 || digit > 7)
6000254     {
6000255         errset (EINVAL);      // Invalid argument.
6000256         perror (NULL);
6000257         return;
6000258     }
6000259     mask = mask * 8 + digit;
6000260 }
6000261 //
6000262 // Set the umask and return.
6000263 //
6000264 umask (mask);
6000265 return;
6000266 }
```

96.1.36 applic/t_fcntl.c



Si veda la sezione [86.25](#).

```
6010001 #include <stdio.h>
6010002 #include <unistd.h>
6010003 #include <stdlib.h>
6010004 #include <sys/wait.h>
6010005 #include <fcntl.h>
6010006 //-----
6010007 int
6010008 main (void)
6010009 {
6010010     int status;
6010011
6010012
6010013     printf ("opzione O_NONBLOCK=%08x\n", O_NONBLOCK);
6010014
6010015     fcntl (STDIN_FILENO, F_SETFL, O_NONBLOCK);
6010016
6010017     status = fcntl (STDIN_FILENO, F_GETFL);
6010018 }
```

```
6010019
6010020
6010021     printf ("fcntl (STDIN_FILENO, F_GETFL) == %08x\n",
6010022             status);
6010023
6010024     return (0);
6010025
6010026 }
```

96.1.37 applic/t_fifo.c

Si veda la sezione [86.25](#).

```
6020001 #include <stdio.h>
6020002 #include <unistd.h>
6020003 #include <stdlib.h>
6020004 #include <sys/wait.h>
6020005 #include <signal.h>
6020006 #include <sys/wait.h>
6020007 #include <string.h>
6020008 #include <fcntl.h>
6020009 #include <sys/stat.h>
6020010 //-----
6020011 int
6020012 main (void)
6020013 {
6020014     int fd;
6020015     pid_t child;
6020016     char buffer;
6020017     char *message =
6020018         "ciao a tutti voi amici vicini e lontani\n";
6020019     int i;
6020020     size_t size;
6020021     ssize_t written;
6020022     int status;
6020023     //
6020024     //
```

```
6020025 //
6020026 unlink ("/tmp/fifo");
6020027 //
6020028 status =
6020029     mknod ("/tmp/fifo", S_IFIFO | S_IRUSR | S_IWUSR, 0);
6020030 if (status != 0)
6020031     {
6020032         perror ("mknod fifo");
6020033         exit (EXIT_FAILURE);
6020034     }
6020035 //
6020036 //
6020037 //
6020038 child = fork ();
6020039 if (child == -1)
6020040     {
6020041         perror ("fork");
6020042         exit (EXIT_FAILURE);
6020043     }
6020044 //
6020045 //
6020046 //
6020047 if (child == 0)
6020048     {
6020049         //
6020050         // This is the child and it have to read the
6020051         // fifo.
6020052         //
6020053         fd = open ("/tmp/fifo", O_RDONLY);
6020054         if (fd < 0)
6020055             {
6020056                 perror ("fifo read open");
6020057                 exit (EXIT_FAILURE);
6020058             }
6020059         //
6020060         // Read one byte at the time, as long as there
6020061         // is
```

```
6020062     // something to read.
6020063     //
6020064     while (read (fd, &buffer, 1) > 0)
6020065     {
6020066         write (STDOUT_FILENO, &buffer, 1);
6020067     }
6020068     //
6020069     // Close the fifo and exit the child.
6020070     //
6020071     close (fd);
6020072     //
6020073     exit (EXIT_SUCCESS);
6020074 }
6020075 else
6020076 {
6020077     //
6020078     // This is the parent process and it writes to
6020079     // the FIFO.
6020080     //
6020081     fd = open ("/tmp/fifo", O_WRONLY);
6020082     if (fd < 0)
6020083     {
6020084         perror ("fifo write open");
6020085         exit (EXIT_FAILURE);
6020086     }
6020087     //
6020088     while (1)
6020089     {
6020090         for (i = 0, written = 0, size =
6020091             strlen (message); i < strlen (message);
6020092             i += written, size -= written)
6020093         {
6020094             written = write (fd, &message[i], size);
6020095             if (written < 0)
6020096             {
6020097                 perror ("pipe");
6020098                 close (fd);
```

```
602009          wait (NULL); // Wait for child.
602010          exit (EXIT_FAILURE);
602011      }
602012  }
602013  }
602014      close (fd); // Reader will see EOF */
602015      wait (NULL); // Wait for child */
602016      exit (EXIT_SUCCESS);
602017  }
602018  //
602019      return (0);
602010 }
```

96.1.38 applic/t_grp.c



Si veda la sezione [86.25](#).

```
6030001 #include <stdio.h>
6030002 #include <grp.h>
6030003 #include <pwd.h>
6030004 #include <unistd.h>
6030005 #include <stdlib.h>
6030006 #include <sys/wait.h>
6030007 #include <signal.h>
6030008 #include <sys/wait.h>
6030009 #include <string.h>
6030010 #include <fcntl.h>
6030011 #include <sys/stat.h>
6030012 //-----
6030013 int
6030014 main (void)
6030015 {
6030016     struct passwd *pw;
6030017     struct group *gr;
6030018     int i;
6030019
6030020     pw = getpwuid ((uid_t) 1001);
```



```
6030021
6030022     if (pw == NULL)
6030023     {
6030024         perror (NULL);
6030025         exit (0);
6030026     }
6030027
6030028     printf ("%s:%s:%i:%i:\n", pw->pw_name, pw->pw_passwd,
6030029            pw->pw_uid, pw->pw_gid);
6030030
6030031     gr = getgrgid ((gid_t) 233);
6030032
6030033     if (gr == NULL)
6030034     {
6030035         perror (NULL);
6030036         exit (0);
6030037     }
6030038
6030039     printf ("%s:%s:%i:", gr->gr_name, gr->gr_passwd,
6030040            gr->gr_gid);
6030041     for (i = 0; i < 32 && gr->gr_mem[i] != NULL; i++)
6030042     {
6030043         printf ("%s,", gr->gr_mem[i]);
6030044     }
6030045
6030046     return (0);
6030047 }
```

96.1.39 applic/t_nc.c

Si veda la sezione [86.25](#).

```
6040001 #include <sys/stat.h>
6040002 #include <sys/types.h>
6040003 #include <unistd.h>
6040004 #include <stdlib.h>
6040005 #include <fcntl.h>
```



```
6040006 #include <errno.h>
6040007 #include <signal.h>
6040008 #include <stdio.h>
6040009 #include <string.h>
6040010 #include <limits.h>
6040011 #include <libgen.h>
6040012 #include <arpa/inet.h>
6040013 #include <sys/socket.h>
6040014 #include <stdint.h>
6040015 #include <stdbool.h>
6040016 #include <fcntl.h>
6040017 //-----
6040018 static void usage (void);
6040019 char buffer[BUFSIZ];
6040020 //-----
6040021 int
6040022 main (int argc, char *argv[], char *envp[])
6040023 {
6040024     bool option_l = 0;
6040025     bool option_u = 0;
6040026     int opt;
6040027 //extern char *optarg;           // not used.
6040028     extern int optind;
6040029     extern int optopt;
6040030     //
6040031     int status;
6040032     int sfdn;
6040033     int sfdn2;
6040034     struct sockaddr_in sa_local;
6040035     struct sockaddr_in sa_remote;
6040036     socklen_t sa_remote_size = sizeof (struct sockaddr_in);
6040037     ssize_t read_size;
6040038     ssize_t sent_size;
6040039     ssize_t recv_size;
6040040     char *addr = NULL;
6040041     char *port = NULL;
6040042     bool can_rx = 1;
```

```
6040043     bool can_tx = 1;
6040044     //
6040045     // Check for options.
6040046     //
6040047     while ((opt = getopt (argc, argv, ":ul")) != -1)
6040048     {
6040049         switch (opt)
6040050         {
6040051             case 'l':
6040052                 option_l = 1;
6040053                 break;
6040054             case 'u':
6040055                 option_u = 1;
6040056                 break;
6040057             case '?':
6040058                 fprintf (stderr, "Unknown option -%c.\n", optopt);
6040059                 usage ();
6040060                 return (1);
6040061                 break;
6040062             case ':':
6040063                 fprintf (stderr,
6040064                         "Missing argument for option -%c\n",
6040065                         optopt);
6040066                 usage ();
6040067                 return (1);
6040068                 break;
6040069             default:
6040070                 fprintf (stderr,
6040071                         "Getopt problem: unknown option %c\n",
6040072                         opt);
6040073                 usage ();
6040074                 return (1);
6040075         }
6040076     }
6040077     //
6040078     // Arguments.
6040079     //
```

```
6040080     if (optind == (argc - 2))
6040081     {
6040082         //
6040083         // There are exactly two arguments: destination
6040084         // address and port.
6040085         //
6040086         addr = argv[argc - 2];
6040087         port = argv[argc - 1];
6040088     }
6040089     else
6040090     {
6040091         //
6040092         // Arguments wrong!
6040093         //
6040094         usage ();
6040095         return (2);
6040096     }
6040097     //
6040098     // Set the local or the remote address.
6040099     //
6040100     if (option_l)
6040101     {
6040102         //
6040103         // Address and port are local.
6040104         //
6040105         sa_local.sin_family = AF_INET;
6040106         sa_local.sin_port = htons (atoi (port));
6040107         inet_pton (AF_INET, addr, &sa_local.sin_addr.s_addr);
6040108     }
6040109     else
6040110     {
6040111         //
6040112         // Address and port are remote.
6040113         //
6040114         sa_remote.sin_family = AF_INET;
6040115         sa_remote.sin_port = htons (atoi (port));
6040116         inet_pton (AF_INET, addr, &sa_remote.sin_addr.s_addr);
```

```
6040117     }
6040118     //
6040119     // Open the socket.
6040120     //
6040121     if (option_u)
6040122     {
6040123         sfdn = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
6040124     }
6040125     else
6040126     {
6040127         sfdn = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
6040128     }
6040129     if (sfdn < 0)
6040130     {
6040131         perror (NULL);
6040132         return (3);
6040133     }
6040134     //
6040135     // Set it listening or connect.
6040136     //
6040137     if (option_l)
6040138     {
6040139         //
6040140         // Bind the local 'sa' location.
6040141         //
6040142         status =
6040143             bind (sfdn, (struct sockaddr *) &sa_local,
6040144                 sizeof (sa_local));
6040145         if (status < 0)
6040146         {
6040147             perror (NULL);
6040148             close (sfdn);
6040149             return (4);
6040150         }
6040151         //
6040152         // Listen (TCP) or wait the first packet (UDP).
6040153         //
```

```
6040154     if (option_u)
6040155     {
6040156         //
6040157         // Instead of listening, we use the function
6040158         // 'recvfrom()',
6040159         // to get the remote address and port.
6040160         //
6040161         recv_size =
6040162             recvfrom (sfdn, &buffer,
6040163                     (size_t) BUFSIZ - 1, 0,
6040164                     (struct sockaddr *) &sa_remote,
6040165                     &sa_remote_size);
6040166         if (recv_size < 0)
6040167         {
6040168             perror (NULL);
6040169             close (sfdn);
6040170             return (4);
6040171         }
6040172         //
6040173         // Now connect the remote destination
6040174         //
6040175         status =
6040176             connect (sfdn,
6040177                    (struct sockaddr *) &sa_remote,
6040178                    sizeof (sa_remote));
6040179         if (status < 0)
6040180         {
6040181             perror (NULL);
6040182             close (sfdn);
6040183             return (7);
6040184         }
6040185         //
6040186         // And show what was received as a first
6040187         // packet.
6040188         //
6040189         buffer[recv_size] = 0;
6040190         printf ("%s", buffer);
```

```
6040191     }
6040192     else
6040193     {
6040194         //
6040195         // TCP: listen.
6040196         //
6040197         status = listen (sfdn, 1);
6040198         if (status < 0)
6040199         {
6040200             perror (NULL);
6040201             close (sfdn);
6040202             return (5);
6040203         }
6040204         //
6040205         // Accept.
6040206         //
6040207         sfdn2 =
6040208             accept (sfdn,
6040209                 (struct sockaddr *) &sa_remote,
6040210                 &sa_remote_size);
6040211         if (sfdn2 < 0)
6040212         {
6040213             perror (NULL);
6040214             close (sfdn);
6040215             return (6);
6040216         }
6040217         //
6040218         // Close listening socket.
6040219         //
6040220         close (sfdn);
6040221         //
6040222         // Variable 'sfdn' will be the new socket.
6040223         //
6040224         sfdn = sfdn2;
6040225     }
6040226 }
6040227 else
```

```
6040228     {
6040229         //
6040230         // Connect the remote destination.
6040231         //
6040232         status =
6040233             connect (sfdn, (struct sockaddr *) &sa_remote,
6040234                     sizeof (sa_remote));
6040235         if (status < 0)
6040236             {
6040237                 perror (NULL);
6040238                 close (sfdn);
6040239                 return (7);
6040240             }
6040241     }
6040242     //
6040243     // Define the standard input non blocking.
6040244     //
6040245     status = fcntl (STDIN_FILENO, F_SETFL, O_NONBLOCK);
6040246     if (status < 0)
6040247         {
6040248             perror (NULL);
6040249             return (8);
6040250         }
6040251     //
6040252     // Will read from the remote and show to the screen.
6040253     //
6040254     while (can_rx || can_tx)
6040255         {
6040256             if (can_rx)
6040257                 {
6040258                     recv_size =
6040259                         recv (sfdn, &buffer, (size_t) BUFSIZ - 1, 0);
6040260                     // recv_size = read (sfdn, &buffer, (size_t) BUFSIZ-1);
6040261                     if (recv_size < 0)
6040262                         {
6040263                             if (errno == EAGAIN || errno == EWOULDBLOCK)
6040264                                 {
```



```
6040265         ;
6040266     }
6040267     else
6040268     {
6040269         perror (NULL);
6040270         close (sfdn);
6040271         return (10);
6040272     }
6040273 }
6040274 else if (recv_size == 0)
6040275 {
6040276     //
6040277     // End of stream.
6040278     //
6040279     can_rx = 0;
6040280     printf ("--end of receive stream--\n");
6040281 }
6040282 else
6040283 {
6040284     buffer[recv_size] = 0;
6040285     printf ("%s", buffer);
6040286 }
6040287 }
6040288 if (can_tx)
6040289 {
6040290     read_size = read (STDIN_FILENO, buffer, BUFSIZ);
6040291     if (read_size < 0)
6040292     {
6040293         if (errno == EAGAIN || errno == EWOULDBLOCK)
6040294         {
6040295             ;
6040296         }
6040297     else
6040298     {
6040299         perror (NULL);
6040300         close (sfdn);
6040301         return (11);
```

```
6040302         }
6040303     }
6040304     else if (read_size == 0)
6040305     {
6040306         //
6040307         // End of input.
6040308         //
6040309         printf ("--closing send stream--\n");
6040310         can_tx = 0;
6040311     }
6040312     else
6040313     {
6040314         //
6040315         // Send it.
6040316         //
6040317         sent_size =
6040318             send (sfdn, &buffer, (size_t) read_size, 0);
6040319         if (sent_size < 0)
6040320         {
6040321             if (errno == EAGAIN
6040322                 || errno == EWOULDBLOCK)
6040323             {
6040324                 ;
6040325             }
6040326             else
6040327             {
6040328                 perror (NULL);
6040329                 close (sfdn);
6040330                 return (12);
6040331             }
6040332         }
6040333     }
6040334 }
6040335 }
6040336 //
6040337 // All done.
6040338 //
```

```

6040339     close (sfdn);
6040340     return (0);
6040341 }
6040342
6040343 //-----
6040344 static void
6040345 usage (void)
6040346 {
6040347     fprintf (stderr,
6040348             "os32 netcat usage:\n"
6040349             "\n"
6040350             "nc [-u] [-l] ADDRESS PORT\n"
6040351             "\n"
6040352             "-u      Use UDP protocol instead of TCP.\n"
6040353             "-l      Listen for incoming connection requests.\n"
6040354             "ADDRESS IPv4 numeric address; if option -l is used, the"
6040355             "      is the local address, otherwise it is the remote"
6040356             "      address.\n"
6040357             "PORT   TCP or UDP port; if option -l is used, this is"
6040358             "      local address, otherwise it is the remote\n"
6040359             "      address.\n");
6040360 }

```

96.1.40 applic/t_ping2.c

Si veda la sezione [86.25](#).

```

6050001 #include <stdio.h>
6050002 #include <sys/types.h>
6050003 //#include <arpa/inet.h>
6050004 #include <sys/socket.h>
6050005 #include <unistd.h>
6050006 #include <errno.h>
6050007 //-----
6050008 int
6050009 main (void)
6050010 {

```



```
6050011 int i;
6050012 int sfdn;
6050013 struct sockaddr_in sa;
6050014 ssize_t spediti;
6050015 ssize_t ricevuti;
6050016 int status;
6050017 uint8_t buffer[100];
6050018 uint8_t packet[] =
6050019     { 0x45, 0x00, 0x00, 0x22, 0x00, 0x00, 0x40, 0x00,
6050020       0x40, 0x01, 0x3c, 0xd9, 0x7f, 0x00, 0x00, 0x01,
6050021       0x7f, 0x00, 0x00, 0x01, 'c', 'i', 'a', 'o', ' ',
6050022       'a', 'm', 'o', 'r', 'e', ' ', 'm', 'i', 'o'
6050023 };
6050024
6050025 sa.sin_family = AF_INET;
6050026 sa.sin_port = 0;
6050027 // sa.sin_addr.s_addr=htonl (0xAC15FEFE); //172.21.254.254
6050028 sa.sin_addr.s_addr = htonl (0xAC150B12); // 172.21.11.18
6050029
6050030 errno = 0;
6050031 sfdn = socket (AF_INET, SOCK_RAW, IPPROTO_ICMP);
6050032 perror (NULL);
6050033
6050034 errno = 0;
6050035 status =
6050036     connect (sfdn, (struct sockaddr *) &sa, sizeof (sa));
6050037 perror (NULL);
6050038
6050039 errno = 0;
6050040 spediti = send (sfdn, packet, 34, 0);
6050041 printf ("scritti %i\n", spediti);
6050042 perror (NULL);
6050043
6050044 ricevuti = 10;
6050045 while (ricevuti > 0)
6050046     {
6050047         errno = 0;
```

```
6050048     ricevuti = recv (sfdn, buffer, (size_t) 30, 0);
6050049     printf ("ricevuti=%i\n", (int) ricevuti);
6050050     perror (NULL);
6050051
6050052     if (ricevuti > 0)
6050053     {
6050054         for (i = 0; i < ricevuti; i++)
6050055         {
6050056             printf ("%02x", (unsigned int) buffer[i]);
6050057         }
6050058     }
6050059 }
6050060
6050061 close (sfdn);
6050062 return (0);
6050063 }
```

96.1.41 applic/t_pipe.c

Si veda la sezione [86.25](#).

```
6060001 #include <stdio.h>
6060002 #include <unistd.h>
6060003 #include <stdlib.h>
6060004 #include <sys/wait.h>
6060005 #include <signal.h>
6060006 #include <sys/wait.h>
6060007 #include <stdio.h>
6060008 #include <stdlib.h>
6060009 #include <string.h>
6060010 //-----
6060011 int
6060012 main (void)
6060013 {
6060014     int pipefd[2];
6060015     pid_t child;
6060016     char buffer;
```



```
6060017 char *message =
6060018     "ciao a tutti voi amici vicini e lontani\n";
6060019 int i;
6060020 size_t size;
6060021 ssize_t written;
6060022 //
6060023 //
6060024 //
6060025 if (pipe (pipefd) == -1)
6060026     {
6060027     perror ("pipe");
6060028     exit (EXIT_FAILURE);
6060029     }
6060030 //
6060031 //
6060032 //
6060033 child = fork ();
6060034 if (child == -1)
6060035     {
6060036     perror ("fork");
6060037     exit (EXIT_FAILURE);
6060038     }
6060039 //
6060040 //
6060041 //
6060042 if (child == 0)
6060043     {
6060044     //
6060045     // This is the child and it have to read the
6060046     // pipe:
6060047     // close the write end of the pipe.
6060048     //
6060049     close (pipefd[1]);
6060050     //
6060051     // Read one byte at the time, as long as there
6060052     // is
6060053     // something to read.
```

```
6060054 //
6060055 while (read (pipefd[0], &buffer, 1) > 0)
6060056 {
6060057     write (STDOUT_FILENO, &buffer, 1);
6060058 }
6060059 //
6060060 // Close the pipe and exit the child.
6060061 //
6060062 close (pipefd[0]);
6060063 //
6060064 exit (EXIT_SUCCESS);
6060065 }
6060066 else
6060067 {
6060068 //
6060069 // This is the parent process, and the read end
6060070 // of
6060071 // pipe is closed.
6060072 //
6060073 close (pipefd[0]);
6060074 //
6060075 while (1)
6060076 {
6060077     for (i = 0, written = 0, size =
6060078         strlen (message); i < strlen (message);
6060079         i += written, size -= written)
6060080     {
6060081         written =
6060082             write (pipefd[1], &message[i], size);
6060083         if (written < 0)
6060084             {
6060085                 perror ("pipe");
6060086                 close (pipefd[1]);
6060087                 wait (NULL); // Wait for child.
6060088                 exit (EXIT_FAILURE);
6060089             }
6060090     }
}
```

```
6060091     }
6060092     close (pipefd[1]);           /* Reader will see EOF */
6060093     wait (NULL);                /* Wait for child */
6060094     exit (EXIT_SUCCESS);
6060095 }
6060096 //
6060097 return (0);
6060098 }
```

96.1.42 applic/t_read.c

«

Si veda la sezione [86.25](#).

```
6070001 #include <sys/stat.h>
6070002 #include <sys/types.h>
6070003 #include <unistd.h>
6070004 #include <stdlib.h>
6070005 #include <fcntl.h>
6070006 #include <errno.h>
6070007 #include <signal.h>
6070008 #include <stdio.h>
6070009 #include <string.h>
6070010 #include <limits.h>
6070011 #include <libgen.h>
6070012 #include <arpa/inet.h>
6070013 #include <sys/socket.h>
6070014 #include <stdint.h>
6070015 #include <stdbool.h>
6070016 #include <fcntl.h>
6070017
6070018 char buffer[BUFSIZ];
6070019
6070020 //-----
6070021 int
6070022 main (int argc, char *argv[], char *envp[])
6070023 {
6070024     int status;
```



```
6070025     ssize_t read_size;
6070026
6070027
6070028     //
6070029     // Define the standard input non blocking.
6070030     //
6070031     status = fcntl (STDIN_FILENO, F_SETFL, O_NONBLOCK);
6070032     if (status < 0)
6070033     {
6070034         perror (NULL);
6070035         return (2);
6070036     }
6070037
6070038
6070039     read_size = read (STDIN_FILENO, buffer, BUFSIZ);
6070040     if (read_size < 0)
6070041     {
6070042         if (errno == EAGAIN || errno == EWOULDBLOCK)
6070043         {
6070044             printf ("nulla da leggere per ora\n");
6070045         }
6070046         else
6070047         {
6070048             perror (NULL);
6070049             return (0);
6070050         }
6070051     }
6070052     else
6070053     {
6070054         buffer[read_size] = 0;
6070055         printf ("letto: %s\n", buffer);
6070056     }
6070057     printf ("finito\n");
6070058     return (0);
6070059 }
```

96.1.43 applic/t_ret.c



Si veda la sezione [86.25](#).

```
6080001 #include <stdlib.h>
6080002 //-----
6080003 int
6080004 main (void)
6080005 {
6080006 //    exit (1);
6080007     return (1);
6080008 }
```

96.1.44 applic/t_rx_udp.c



Si veda la sezione [86.25](#).

```
6090001 #include <sys/stat.h>
6090002 #include <sys/types.h>
6090003 #include <unistd.h>
6090004 #include <stdlib.h>
6090005 #include <fcntl.h>
6090006 #include <errno.h>
6090007 #include <signal.h>
6090008 #include <stdio.h>
6090009 #include <string.h>
6090010 #include <limits.h>
6090011 #include <libgen.h>
6090012 #include <arpa/inet.h>
6090013 #include <sys/socket.h>
6090014 #include <stdint.h>
6090015 #include <stdbool.h>
6090016 //-----
6090017 static void usage (void);
6090018 //-----
6090019 int
6090020 main (int argc, char *argv[], char *envp[])
6090021 {
```

```
6090022     int status;
6090023     int sfdn;
6090024     struct sockaddr_in sa_local;
6090025     ssize_t recv_size;
6090026     char buffer[BUFSIZ];
6090027     char *addr = NULL;
6090028     char *port = NULL;
6090029     //
6090030     // Arguments.
6090031     //
6090032     if (argc == 3)
6090033     {
6090034         //
6090035         // There are exactly two arguments: destination
6090036         // address and port.
6090037         //
6090038         addr = argv[1];
6090039         port = argv[2];
6090040     }
6090041     else
6090042     {
6090043         //
6090044         // Arguments wrong!
6090045         //
6090046         usage ();
6090047         return (4);
6090048     }
6090049     //
6090050     // Define the destination 'sa_local'
6090051     //
6090052     sa_local.sin_family = AF_INET;
6090053     sa_local.sin_port = htons (atoi (port));
6090054     inet_pton (AF_INET, addr, &sa_local.sin_addr.s_addr);
6090055     //
6090056     // Open the socket.
6090057     //
6090058     sfdn = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

```
6090059     if (sfdn < 0)
6090060     {
6090061         perror (NULL);
6090062         return (5);
6090063     }
6090064     //
6090065     // Bind the local 'sa' location.
6090066     //
6090067     status = bind (sfdn, (struct sockaddr *) &sa_local,
6090068                   sizeof (sa_local));
6090069     if (status < 0)
6090070     {
6090071         perror (NULL);
6090072         close (sfdn);
6090073         return (7);
6090074     }
6090075     //
6090076     // Will read from the remote and show to the screen.
6090077     //
6090078     while (1)
6090079     {
6090080         recv_size = read (sfdn, &buffer, (size_t) BUFSIZ - 1);
6090081         if (recv_size < 0)
6090082         {
6090083             perror (NULL);
6090084             close (sfdn);
6090085             return (10);
6090086         }
6090087         buffer[recv_size] = 0;
6090088         printf ("%s", buffer);
6090089     }
6090090     //
6090091     // All done.
6090092     //
6090093     return (0);
6090094 }
6090095
```

```
6090096 //-----
6090097 static void
6090098 usage (void)
6090099 {
6090100     fprintf (stderr, "Usage: rx_udp LOCAL_ADDR LOCAL_PORT\n");
6090101 }
```

96.1.45 applic/t_scr.c

Si veda la sezione [86.25](#).

```
6100001 #include <unistd.h>
6100002 #include <stdio.h>
6100003 #include <fcntl.h>
6100004 #include <unistd.h>
6100005 #include <stdlib.h>
6100006 //-----
6100007 int
6100008 main (int argc, char *argv[], char *envp[])
6100009 {
6100010     FILE *screen;
6100011     int status;
6100012
6100013     screen = fopen ("/dev/tty", "w");
6100014     if (screen == NULL)
6100015     {
6100016         printf ("[%s] Cannot open \"/dev/tty\" ", argv[0]);
6100017         perror (NULL);
6100018         exit (0);
6100019     }
6100020
6100021     status = fseek (screen, (long) 1000, SEEK_SET);
6100022
6100023     fprintf (screen, "ciao status: %i ciao", status);
6100024     perror (NULL);
6100025
6100026     fclose (screen);
```

```
6100027     return (0);
6100028 }
```

96.1.46 applic/t_setjmp.c



Si veda la sezione [86.25](#).

```
6110001 #include <stdio.h>
6110002 #include <setjmp.h>
6110003 //-----
6110004
6110005 jmp_buf env;
6110006
6110007 void
6110008 prova3 (void)
6110009 {
6110010     printf ("funzione prova3\n");
6110011     longjmp (env, 1);
6110012     printf ("funzione prova3 post\n");
6110013 }
6110014
6110015 void
6110016 prova2 (void)
6110017 {
6110018     printf ("funzione prova2\n");
6110019     prova3 ();
6110020     printf ("funzione prova2 post\n");
6110021 }
6110022
6110023 void
6110024 prova1 (void)
6110025 {
6110026     printf ("funzione prova1\n");
6110027     prova2 ();
6110028     printf ("funzione prova1 post\n");
6110029 }
6110030
```

```
6110031 int
6110032 main (int argc, char *argv[], char *envp[])
6110033 {
6110034     int val;
6110035     //
6110036     printf ("prima\n");
6110037     //
6110038     val = setjmp (env);
6110039     //
6110040     printf ("dopo setjmp val=%i\n", val);
6110041     //
6110042     if (val != 0)
6110043         return (0);
6110044
6110045     proval ();
6110046
6110047     return (0);
6110048 }
```

96.1.47 applic/t_sig.c

Si veda la sezione [86.25](#).

```
6120001 #include <stdio.h>
6120002 #include <unistd.h>
6120003 #include <stdlib.h>
6120004 #include <sys/wait.h>
6120005 #include <signal.h>
6120006 //-----
6120007 void
6120008 signal_handler (int signal)
6120009 {
6120010     printf ("Hello! I have caught the signal %i.\n", signal);
6120011 }
6120012
6120013 //-----
6120014 int
```

```
6120015 main (void)
6120016 {
6120017     signal (SIGHUP, signal_handler);
6120018     signal (SIGINT, signal_handler);
6120019     signal (SIGQUIT, signal_handler);
6120020     signal (SIGILL, signal_handler);
6120021     signal (SIGABRT, signal_handler);
6120022     signal (SIGFPE, signal_handler);
6120023     signal (SIGKILL, signal_handler);
6120024     signal (SIGSEGV, signal_handler);
6120025     signal (SIGPIPE, signal_handler);
6120026     signal (SIGALRM, signal_handler);
6120027     signal (SIGTERM, signal_handler);
6120028     signal (SIGSTOP, signal_handler);
6120029     signal (SIGTSTP, signal_handler);
6120030     signal (SIGCONT, signal_handler);
6120031     signal (SIGCHLD, signal_handler);
6120032     signal (SIGTTIN, signal_handler);
6120033     signal (SIGTTOU, signal_handler);
6120034     signal (SIGUSR1, signal_handler);
6120035     signal (SIGUSR2, signal_handler);
6120036     //
6120037     while (1)
6120038     {
6120039         sleep (1);
6120040         printf ("ciao!\n");
6120041     }
6120042     //
6120043     return (0);
6120044 }
```

96.1.48 applic/t_sig2.c



Si veda la sezione [86.25](#).

```
6130001 #include <stdio.h>
6130002 #include <unistd.h>
```



```
6130003 #include <stdlib.h>
6130004 #include <sys/wait.h>
6130005 #include <signal.h>
6130006 //-----
6130007 void
6130008 signal_handler (int signal)
6130009 {
6130010     printf ("Hello! I have caught the signal %i.\n", signal);
6130011 }
6130012
6130013 //-----
6130014 int
6130015 main (void)
6130016 {
6130017     //
6130018     while (1)
6130019     {
6130020         signal (SIGHUP, signal_handler);
6130021         signal (SIGINT, signal_handler);
6130022         signal (SIGQUIT, signal_handler);
6130023         signal (SIGILL, signal_handler);
6130024         signal (SIGABRT, signal_handler);
6130025         signal (SIGFPE, signal_handler);
6130026         signal (SIGKILL, signal_handler);
6130027         signal (SIGSEGV, signal_handler);
6130028         signal (SIGPIPE, signal_handler);
6130029         signal (SIGALRM, signal_handler);
6130030         signal (SIGTERM, signal_handler);
6130031         signal (SIGSTOP, signal_handler);
6130032         signal (SIGTSTP, signal_handler);
6130033         signal (SIGCONT, signal_handler);
6130034         signal (SIGCHLD, signal_handler);
6130035         signal (SIGTTIN, signal_handler);
6130036         signal (SIGTTOU, signal_handler);
6130037         signal (SIGUSR1, signal_handler);
6130038         signal (SIGUSR2, signal_handler);
6130039         printf ("ciao!\n");
```

```
6130040     sleep (1);
6130041     }
6130042     //
6130043     return (0);
6130044 }
```

96.1.49 applic/t_tx_tcp.c

<<

Si veda la sezione [86.25](#).

```
6140001 #include <sys/stat.h>
6140002 #include <sys/types.h>
6140003 #include <unistd.h>
6140004 #include <stdlib.h>
6140005 #include <fcntl.h>
6140006 #include <errno.h>
6140007 #include <signal.h>
6140008 #include <stdio.h>
6140009 #include <string.h>
6140010 #include <limits.h>
6140011 #include <libgen.h>
6140012 #include <arpa/inet.h>
6140013 #include <sys/socket.h>
6140014 #include <stdint.h>
6140015 #include <stdbool.h>
6140016 //-----
6140017 static void usage (void);
6140018 //-----
6140019 int
6140020 main (int argc, char *argv[], char *envp[])
6140021 {
6140022     int status;
6140023     int sfdn;
6140024     struct sockaddr_in sa;
6140025     ssize_t read_size;
6140026     ssize_t sent_size;
6140027     char buffer[BUFSIZ];
```

```
6140028 char *addr = NULL;
6140029 char *port = NULL;
6140030 //
6140031 // Arguments.
6140032 //
6140033 if (argc == 3)
6140034 {
6140035     //
6140036     // There are exactly two arguments: destination
6140037     // address and port.
6140038     //
6140039     addr = argv[1];
6140040     port = argv[2];
6140041 }
6140042 else
6140043 {
6140044     //
6140045     // Arguments wrong!
6140046     //
6140047     usage ();
6140048     return (4);
6140049 }
6140050 //
6140051 // Define the destination 'sa'
6140052 //
6140053 sa.sin_family = AF_INET;
6140054 sa.sin_port = htons (atoi (port));
6140055 inet_pton (AF_INET, addr, &sa.sin_addr.s_addr);
6140056 //
6140057 //
6140058 // Open the socket.
6140059 //
6140060 sfdn = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
6140061 if (sfdn < 0)
6140062 {
6140063     perror (NULL);
6140064     return (5);
```

```
6140065     }
6140066     //
6140067     // Connect the 'sa' destination
6140068     //
6140069     status =
6140070         connect (sfdn, (struct sockaddr *) &sa, sizeof (sa));
6140071     if (status < 0)
6140072     {
6140073         perror (NULL);
6140074         close (sfdn);
6140075         return (7);
6140076     }
6140077     //
6140078     // Will read from the standard input and send to the
6140079     // other
6140080     // side.
6140081     //
6140082     while (1)
6140083     {
6140084         read_size = read (STDIN_FILENO, buffer, BUFSIZ);
6140085         if (read_size < 0)
6140086         {
6140087             perror (NULL);
6140088             close (sfdn);
6140089             return (8);
6140090         }
6140091         if (read_size == 0)
6140092         {
6140093             close (sfdn);
6140094             return (0);
6140095         }
6140096         //
6140097         // Verify the 'stop' command.
6140098         //
6140099         if (strncmp (buffer, "stop\n", read_size) == 0)
6140100         {
6140101             printf ("closing send...\n");
```

```
6140102         close (sfdn);
6140103         return (0);
6140104     }
6140105     //
6140106     sent_size =
6140107         send (sfdn, &buffer, (size_t) read_size, 0);
6140108     if (sent_size < 0)
6140109     {
6140110         perror (NULL);
6140111         close (sfdn);
6140112         return (9);
6140113     }
6140114     printf ("sent %i bytes\n", (int) sent_size);
6140115 }
6140116 //
6140117 // All done.
6140118 //
6140119 return (0);
6140120 }
6140121
6140122 //-----
6140123 static void
6140124 usage (void)
6140125 {
6140126     fprintf (stderr, "Usage: tx_tcp DEST_ADDR DEST_PORT\n");
6140127 }
```

96.1.50 applic/t_tx_udp.c

Si veda la sezione [86.25](#).

```
6150001 #include <sys/stat.h>
6150002 #include <sys/types.h>
6150003 #include <unistd.h>
6150004 #include <stdlib.h>
6150005 #include <fcntl.h>
6150006 #include <errno.h>
```

```
6150007 #include <signal.h>
6150008 #include <stdio.h>
6150009 #include <string.h>
6150010 #include <limits.h>
6150011 #include <libgen.h>
6150012 #include <arpa/inet.h>
6150013 #include <sys/socket.h>
6150014 #include <stdint.h>
6150015 #include <stdbool.h>
6150016 //-----
6150017 static void usage (void);
6150018 //-----
6150019 int
6150020 main (int argc, char *argv[], char *envp[])
6150021 {
6150022     int status;
6150023     int sfdn;
6150024     struct sockaddr_in sa;
6150025     ssize_t read_size;
6150026     ssize_t sent_size;
6150027     char buffer[BUFSIZ];
6150028     char *addr = NULL;
6150029     char *port = NULL;
6150030     //
6150031     // Arguments.
6150032     //
6150033     if (argc == 3)
6150034     {
6150035         //
6150036         // There are exactly two arguments: destination
6150037         // address and port.
6150038         //
6150039         addr = argv[1];
6150040         port = argv[2];
6150041     }
6150042     else
6150043     {
```

```
6150044      //
6150045      // Arguments wrong!
6150046      //
6150047      usage ();
6150048      return (4);
6150049  }
6150050  //
6150051  // Define the destination 'sa'
6150052  //
6150053  sa.sin_family = AF_INET;
6150054  sa.sin_port = htons (atoi (port));
6150055  inet_pton (AF_INET, addr, &sa.sin_addr.s_addr);
6150056  //
6150057  //
6150058  // Open the socket.
6150059  //
6150060  sfdn = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
6150061  if (sfdn < 0)
6150062  {
6150063      perror (NULL);
6150064      return (5);
6150065  }
6150066  //
6150067  // Connect the 'sa' destination
6150068  //
6150069  status =
6150070      connect (sfdn, (struct sockaddr *) &sa, sizeof (sa));
6150071  if (status < 0)
6150072  {
6150073      perror (NULL);
6150074      close (sfdn);
6150075      return (7);
6150076  }
6150077  //
6150078  // Will read from the standard input and send to the
6150079  // other
6150080  // side.
```

```
6150081 //
6150082 while (1)
6150083     {
6150084         read_size = read (STDIN_FILENO, buffer, BUFSIZ);
6150085         if (read_size < 0)
6150086             {
6150087                 perror (NULL);
6150088                 close (sfdn);
6150089                 return (8);
6150090             }
6150091         if (read_size == 0)
6150092             {
6150093                 close (sfdn);
6150094                 return (0);
6150095             }
6150096         //
6150097         sent_size =
6150098             send (sfdn, &buffer, (size_t) read_size, 0);
6150099         if (sent_size < 0)
6150100             {
6150101                 perror (NULL);
6150102                 close (sfdn);
6150103                 return (9);
6150104             }
6150105         printf ("sent %i bytes\n", (int) sent_size);
6150106     }
6150107 //
6150108 // All done.
6150109 //
6150110 return (0);
6150111 }
6150112
6150113 //-----
6150114 static void
6150115 usage (void)
6150116 {
6150117     fprintf (stderr, "Usage: tx_udp DEST_ADDR DEST_PORT\n");
```


6150118

}

96.1.51 applic/touch.c

Si veda la sezione [86.26](#).

```
6160001 #include <fcntl.h>
6160002 #include <sys/stat.h>
6160003 #include <utime.h>
6160004 #include <stddef.h>
6160005 #include <unistd.h>
6160006 #include <errno.h>
6160007 //-----
6160008 static void usage (void);
6160009 //-----
6160010 int
6160011 main (int argc, char *argv[], char *envp[])
6160012 {
6160013     int a;           // Argument index.
6160014     int status;
6160015     struct stat file_status;
6160016     //
6160017     // No options are known, but at least an argument
6160018     // must be given.
6160019     //
6160020     if (argc < 2)
6160021     {
6160022         usage ();
6160023         return (1);
6160024     }
6160025     //
6160026     // Scan arguments.
6160027     //
6160028     for (a = 1; a < argc; a++)
6160029     {
6160030         //
6160031         // Verify if the file exists, through the return
```

```
6160032 // value of
6160033 // 'stat()'. No other checks are made.
6160034 //
6160035 if (stat (argv[a], &file_status) == 0)
6160036 {
6160037 //
6160038 // File exists: should be updated the times.
6160039 //
6160040 status = utime (argv[a], NULL);
6160041 if (status != 0)
6160042 {
6160043 perror (NULL);
6160044 return (2);
6160045 }
6160046 }
6160047 else
6160048 {
6160049 //
6160050 // File does not exist: should be created.
6160051 //
6160052 status =
6160053 open (argv[a],
6160054 O_WRONLY | O_CREAT | O_TRUNC, 0666);
6160055 //
6160056 if (status >= 0)
6160057 {
6160058 //
6160059 // Here, the variable 'status' is the
6160060 // file
6160061 // descriptor to be closed.
6160062 //
6160063 status = close (status);
6160064 if (status != 0)
6160065 {
6160066 perror (NULL);
6160067 return (3);
6160068 }
```

```
6160069         }
6160070     else
6160071     {
6160072         perror (NULL);
6160073         return (4);
6160074     }
6160075 }
6160076 }
6160077 return (0);
6160078 }
6160079
6160080 //-----
6160081 static void
6160082 usage (void)
6160083 {
6160084     fprintf (stderr, "Usage: touch FILE...\n");
6160085 }
```

96.1.52 applic/tty.c

Si veda la sezione [86.27](#).

```
6170001 #include <fcntl.h>
6170002 #include <sys/stat.h>
6170003 #include <utime.h>
6170004 #include <stddef.h>
6170005 #include <unistd.h>
6170006 #include <errno.h>
6170007 #include <sys/os32.h>
6170008 #include <sys/types.h>
6170009 //-----
6170010 static void usage (void);
6170011 //-----
6170012 int
6170013 main (int argc, char *argv[], char *envp[])
6170014 {
6170015     int dev_minor;
```

```
6170016 struct stat file_status;
6170017 //
6170018 // No options and no arguments.
6170019 //
6170020 if (argc > 1)
6170021 {
6170022     usage ();
6170023     return (1);
6170024 }
6170025 //
6170026 // Verify the standard input.
6170027 //
6170028 if (fstat (STDIN_FILENO, &file_status) == 0)
6170029 {
6170030     if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
6170031     {
6170032         dev_minor = minor (file_status.st_rdev);
6170033         //
6170034         // If minor is equal to 0xFF, it is
6170035         // '/dev/console'
6170036         // that is not a controlling terminal, but
6170037         // just
6170038         // a reference for the current virtual
6170039         // console.
6170040         //
6170041         if (dev_minor < 0xFF)
6170042         {
6170043             printf ("/dev/console%i\n", dev_minor);
6170044         }
6170045     }
6170046 }
6170047 else
6170048 {
6170049     perror ("Cannot get standard input file status");
6170050     return (2);
6170051 }
6170052 //
```

```
6170053     return (0);
6170054 }
6170055
6170056 //-----
6170057 static void
6170058 usage (void)
6170059 {
6170060     fprintf (stderr, "Usage: tty\n");
6170061 }
```

96.1.53 applic/umount.c

Si veda la sezione [92.7](#).



```
6180001 #include <unistd.h>
6180002 #include <stdlib.h>
6180003 #include <sys/stat.h>
6180004 #include <sys/types.h>
6180005 #include <fcntl.h>
6180006 #include <errno.h>
6180007 #include <signal.h>
6180008 #include <stdio.h>
6180009 #include <sys/wait.h>
6180010 #include <stdio.h>
6180011 #include <string.h>
6180012 #include <limits.h>
6180013 #include <sys/os32.h>
6180014 //-----
6180015 static void usage (void);
6180016 //-----
6180017 int
6180018 main (int argc, char *argv[], char *envp[])
6180019 {
6180020     int status;
6180021     //
6180022     // One argument is mandatory.
6180023     //
```

```
6180024     if (argc != 2)
6180025         {
6180026             usage ();
6180027             return (1);
6180028         }
6180029     //
6180030     // System call.
6180031     //
6180032     status = umount (argv[1]);
6180033     if (status != 0)
6180034         {
6180035             perror (argv[1]);
6180036             return (2);
6180037         }
6180038     //
6180039     return (0);
6180040 }
6180041
6180042 //-----
6180043 static void
6180044 usage (void)
6180045 {
6180046     fprintf (stderr, "Usage: umount MOUNT_POINT\n");
6180047 }
```

96.1.54 applic/yes.c



Si veda la sezione [86.28](#).

```
6190001 #include <stdio.h>
6190002 //-----
6190003 int
6190004 main (int argc, char *argv[], char *envp[])
6190005 {
6190006     int i;
6190007     //
6190008     if (argc > 1)
```

```
6190009     {
6190010         while (1)
6190011         {
6190012             printf ("%s", argv[1]);
6190013             for (i = 2; i < argc; i++)
6190014                 {
6190015                     printf (" %s", argv[i]);
6190016                 }
6190017             printf ("\n");
6190018         }
6190019     }
6190020 else
6190021     {
6190022         while (1)
6190023         {
6190024             printf ("y\n");
6190025         }
6190026     }
6190027 return (0);
6190028 }
```



Indice analitico del volume



aaa [467](#) aaa.c [2238](#) abort () [603](#) abort.c [2018](#) abs () [604](#)
abs.c [2019](#) accept () [498](#) accept.c [2131](#) access () [605](#)
access.c [2183](#) *address resolution protocol* [183](#) addr_t [252](#)
[333](#) allocated [468](#) allocated.c [2239](#) applic.sep.ld
[980](#) arp [777](#) ARP [183](#) arp.c [2241](#) arp.h [417](#) [795](#) [1552](#)
arp_clean () [418](#) arp_clean.c [1553](#) arp_index () [418](#)
arp_index.c [1554](#) arp_init () [418](#) arp_init.c [1556](#)
arp_print.c [1556](#) arp_public.c [1557](#)
arp_reference () [418](#) arp_reference.c [1557](#)
arp_request () [418](#) arp_request.c [1558](#) arp_rx () [418](#)
arp_rx.c [1560](#) asctime () [617](#) asctime.c [2164](#)
assert () [607](#) assert.h [1800](#) ATA [79](#) ata.h [797](#) [1044](#) ata0
[756](#) ata1 [756](#) ata2 [756](#) ata3 [756](#) ata4 [756](#) ata5 [756](#) ata6
[756](#) ata7 [756](#) ata_cmd_identify_device ()
[366](#) ata_cmd_identify_device.c
[1049](#) ata_cmd_read_sectors ()
[366](#) ata_cmd_read_sectors.c
[1051](#) ata_cmd_write_sectors () [366](#)
ata_cmd_write_sectors.c [1053](#) ata_device () [366](#)
ata_device.c [1055](#) ata_drq () [368](#) ata_drq.c [1057](#)
ata_init () [366](#) ata_init.c [1059](#) ata_lba28 () [368](#)
ata_lba28.c [1066](#) ata_public.c [1067](#) ata_rdy () [368](#)
ata_rdy.c [1068](#) ata_read_sector () [369](#) ata_reset ()
[366](#) ata_reset.c [1070](#) ata_sector_t [252](#) [365](#) ata_t [252](#)
[365](#) ata_valid () [366](#) ata_valid.c [1070](#)
ata_write_sector () [369](#) atexit () [608](#) atexit.c [2020](#)
atoi () [610](#) atoi.c [2021](#) atol () [610](#) atol.c [2022](#) avvio

[455](#) [basename\(\)](#) [611](#) [basename.c](#) [1866](#) [bbb.c](#) [2243](#)
[bind\(\)](#) [501](#) [bind.c](#) [2133](#) [blk.h](#) [342](#) [802](#) [1008](#) [blk_ata\(\)](#)
[802](#) [blk_ata.c](#) [342](#) [1010](#) [blk_cache_check\(\)](#) [803](#)
[blk_cache_check.c](#) [1012](#) [blk_cache_init\(\)](#) [805](#)
[blk_cache_init.c](#) [342](#) [1013](#) [blk_cache_read\(\)](#) [806](#)
[blk_cache_read.c](#) [342](#) [1014](#) [blk_cache_save\(\)](#) [806](#)
[blk_cache_save.c](#) [342](#) [1015](#) [blk_cache_t](#) [252](#)
[blk_public.c](#) [1017](#) [bochs](#) [981](#) [brk\(\)](#) [503](#) [brk.c](#) [2184](#)
[build.h](#) [1506](#) [cat](#) [469](#) [cat.c](#) [2244](#) [ccc.c](#) [2246](#) [chdir\(\)](#)
[505](#) [chdir.c](#) [2185](#) [chgrp](#) [470](#) [chgrp.c](#) [2247](#) [chiamata di](#)
[sistema](#) [305](#) [chmod](#) [470](#) [chmod\(\)](#) [507](#) [chmod.c](#) [2148](#) [2250](#)
[chown](#) [471](#) [chown\(\)](#) [510](#) [chown.c](#) [2186](#) [2252](#) [clearerr\(\)](#)
[614](#) [clearerr.c](#) [1903](#) [CLI](#) [24](#) [cli\(\)](#) [275](#) [cli.s](#) [1288](#)
[clock\(\)](#) [512](#) [clock.c](#) [2166](#) [clock_t.h](#) [1801](#) [close\(\)](#) [513](#)
[close.c](#) [2187](#) [closedir\(\)](#) [615](#) [closedir.c](#) [1828](#)
[conclusione](#) [455](#) [condotto](#) [392](#) [connect\(\)](#) [514](#) [connect.c](#)
[2134](#) [console](#) [755](#) [console0](#) [755](#) [console1](#) [755](#) [console2](#)
[755](#) [console3](#) [755](#) [cp](#) [472](#) [cp.c](#) [2255](#) [CPL](#) [16](#) [creat\(\)](#) [616](#)
[creat.c](#) [1850](#) [crt0.mer.s](#) [2261](#) [crt0.s](#) [262](#) [1506](#)
[crt0.sep.s](#) [2265](#) [ctime\(\)](#) [617](#) [ctype.h](#) [1801](#) [date](#) [473](#)
[date.c](#) [2269](#) [dev.h](#) [340](#) [808](#) [1017](#) [dev_ata\(\)](#) [817](#)
[dev_ata.c](#) [1019](#) [DEV_CONSOLE](#) [345](#) [DEV_CONSOLE \$n\$](#) [345](#)
[dev_dm\(\)](#) [816](#) [dev_dm.c](#) [340](#) [1022](#) [DEV_DM \$m\$ \$n\$](#) [345](#)
[dev_io\(\)](#) [340](#) [815](#) [dev_io.c](#) [340](#) [1023](#) [dev_kmem\(\)](#) [818](#)
[dev_kmem.c](#) [340](#) [1024](#) [DEV_KMEM_ARP](#) [345](#)
[DEV_KMEM_FILE](#) [345](#) [DEV_KMEM_INODE](#) [345](#)
[DEV_KMEM_MMP](#) [345](#) [DEV_KMEM_NET](#) [345](#) [DEV_KMEM_PS](#) [345](#)
[DEV_KMEM_ROUTE](#) [345](#) [DEV_KMEM_SB](#) [345](#) [DEV_MEM](#) [345](#)
[dev_mem\(\)](#) [819](#) [dev_mem.c](#) [1031](#) [DEV_NULL](#) [345](#) [DEV_PORT](#)

345 DEV_TTY 345 dev_tty() 822 dev_tty.c 340 1034
DEV_ZERO 345 DIR.c 1827 directory_t 252 dirent.h
1825 dirname() 611 dirname.c 1867 div() 620 div.c
2023 dm.h 823 1037 dm_init.c 1041 dm_public.c 1044
dm_t 364 DPL 16 dup() 516 dup.c 2187 dup2() 516
dup2.c 2188 eccezione 64 ed 475 ed.c 2273 elf-to-os32
982 endgrent() 661 endpwent() 672 environ 773
environ.c 2189 environment.c 2024 errfn 622 errln
622 errno 622 errno.c 1846 errno.h 1835 errset() 622
execl() 634 execl.c 2189 execl_e() 634 execl_e.c 2190
execlp() 634 execlp.c 2191 execv() 634 execv.c 2193
execve() 517 execve.c 2193 execvp() 634 execvp.c
2196 exit() 608 exit.c 2026 fchdir() 505 fchdir.c
2197 fchmod() 507 fchmod.c 2149 fchown() 510
fchown.c 2197 fclose() 637 fclose.c 1903 fcntl()
520 fcntl.c 1850 fcntl.h 1846 fdisk 985 fd_dup() 406
830 fd_dup.c 1159 fd_reference() 406 832
fd_reference.c 1161 fd_t 252 feof() 638 feof.c 1903
ferror() 639 ferror.c 1904 fflush() 640 fflush.c
1904 fgetc() 641 fgetc.c 1905 fgetpos() 643
fgetpos.c 1906 fgets() 644 fgets.c 1906 FILE.c 1902
fileno() 646 fileno.c 1908 file_image_functions
986 file_pipe_make() 834 file_pipe_make.c 1162
file_reference() 397 836 file_reference.c
1163 file_stdio_dev_make() 397 837
file_stdio_dev_make.c 1164 file_t 252 394 fopen()
647 fopen.c 1908 fork() 524 fork.c 2198 format 993
fprintf() 698 fprintf.c 1910 fputc() 651 fputc.c
1911 fputs() 652 fputs.c 1911 fread() 654 fread.c

[1912](#) free() [685](#) free.c [2054](#) freopen() [647](#) freopen.c
[1913](#) fs.h [370](#) [823](#) [1147](#) fscanf() [713](#) fscanf.c [1914](#)
fseek() [655](#) fseek.c [1915](#) fseeko() [655](#) fseeko.c [1915](#)
fsetpos() [643](#) fsetpos.c [1916](#) fstat() [580](#) fstat.c
[2150](#) fs_init() [834](#) fs_init.c [1166](#) fs_public.c [1167](#)
ftell() [657](#) ftell.c [1917](#) ftello() [657](#) ftello.c [1917](#)
fwrite() [659](#) fwrite.c [1917](#) **GDT** [27](#) gdt() [279](#) gdt.c
[1289](#) gdt_load() [279](#) gdt_load.s [1290](#) gdt_print() [279](#)
gdt_print.c [1291](#) gdt_public.c [1292](#) gdt_segment()
[279](#) gdt_segment.c [1292](#) gdt_t [252](#) getc() [641](#)
getchar() [641](#) getchar.c [1918](#) getcwd() [526](#) getcwd.c
[2199](#) getegid() [527](#) getegid.c [2201](#) getenv() [660](#)
getenv.c [2027](#) geteuid() [530](#) geteuid.c [2201](#)
getgid() [527](#) getgid.c [2202](#) getgrent() [661](#)
getgrgid() [664](#) getgrnam() [664](#) getopt() [666](#)
getopt.c [2202](#) getpgrp() [529](#) getpgrp.c [2209](#)
getpid() [529](#) getpid.c [2210](#) getppid() [529](#) getppid.c
[2210](#) getpwent() [672](#) getpwnam() [675](#) getpwuid() [675](#)
gets() [644](#) gets.c [1919](#) getty [778](#) getty.c [2314](#)
getuid() [530](#) getuid.c [2211](#) *global descriptor table* [27](#)
gmtime() [617](#) gmtime.c [2167](#) grent.c [1853](#) group [769](#)
grp.h [1852](#) header_t [252](#) htonl() [613](#) htonl.c [1819](#)
htons() [613](#) htons.c [1819](#) http [779](#) http.c [2317](#)
h_addr_t [252](#) [420](#) ibm_i386.h [274](#) [891](#) [1277](#) **ICMP** [199](#)
icmp.h [897](#) [1565](#) [1870](#) icmp_rx() [428](#) icmp_rx.c [1566](#)
icmp_tx() [428](#) icmp_tx.c [1572](#) icmp_tx_echo() [428](#)
icmp_tx_echo.c [1573](#) icmp_tx_unreachable() [428](#)
icmp_tx_unreachable.c [1574](#) **IDE** [79](#) **IDT** [46](#) idt() [281](#)
idt.c [1294](#) idtr_t [252](#) idt_descriptor() [281](#)

[idt_descriptor.c](#) [1296](#) [idt_irq_remap\(\)](#) [281](#)
[idt_irq_remap.c](#) [1298](#) [idt_load\(\)](#) [281](#) [idt_load.s](#)
[1300](#) [idt_print\(\)](#) [281](#) [idt_print.c](#) [1300](#) [idt_public.c](#)
[1301](#) [idt_t](#) [252](#) [imaxabs\(\)](#) [604](#) [imaxabs.c](#) [1864](#)
[imaxdiv\(\)](#) [620](#) [imaxdiv.c](#) [1865](#) [in.h](#) [1874](#) [INB](#) [22](#) [inet.h](#)
[1818](#) [inet_ntop\(\)](#) [678](#) [inet_ntop.c](#) [1820](#) [inet_pton\(\)](#)
[680](#) [inet_pton.c](#) [1821](#) [init](#) [780](#) [init.c](#) [2333](#) [inittab](#)
[770](#) [inode_alloc\(\)](#) [380 839](#) [inode_alloc.c](#) [1167](#)
[inode_check\(\)](#) [380 840](#) [inode_check.c](#) [1172](#)
[inode_dir_empty\(\)](#) [380 843](#) [inode_dir_empty.c](#) [1175](#)
[inode_file_read\(\)](#) [380 844](#) [inode_file_read.c](#) [1177](#)
[inode_file_write\(\)](#) [380 846](#) [inode_file_write.c](#)
[1181](#) [inode_free\(\)](#) [380 848](#) [inode_free.c](#) [1184](#)
[inode_fzones_read\(\)](#) [380 849](#) [inode_fzones_read.c](#)
[1185](#) [inode_fzones_write\(\)](#) [380 849](#)
[inode_fzones_write.c](#) [1187](#) [inode_get\(\)](#) [380 851](#)
[inode_get.c](#) [1189](#) [inode_pipe_make\(\)](#) [380 853](#)
[inode_pipe_make.c](#) [1195](#) [inode_pipe_read\(\)](#) [380 854](#)
[inode_pipe_read.c](#) [1197](#) [inode_pipe_write\(\)](#) [380 856](#)
[inode_pipe_write.c](#) [1200](#) [inode_print\(\)](#) [380 858](#)
[inode_print.c](#) [1203](#) [inode_put\(\)](#) [380 858](#)
[inode_put.c](#) [1206](#) [inode_reference\(\)](#) [380 860](#)
[inode_reference.c](#) [1208](#) [inode_save\(\)](#) [380 862](#)
[inode_save.c](#) [1211](#) [inode_stdio_dev_make\(\)](#) [380 863](#)
[inode_stdio_dev_make.c](#) [1213](#) [inode_t](#) [252 377](#)
[inode_truncate\(\)](#) [380 864](#) [inode_truncate.c](#) [1215](#)
[inode_zone\(\)](#) [380 865](#) [inode_zone.c](#) [1219](#)
[input_line\(\)](#) [681](#) [input_line.c](#) [2112](#) [interfaccia di rete](#)
[132](#) [interrupt descriptor table](#) [46](#) [interruzione](#) [53](#) [inttypes.h](#)

1857 in_16() 274 in_8() 274 ip.h 420 897 1575 1877
ipconfig 781 ipconfig() 531 ipconfig.c 2116 2340
IPv4 182 189 ip_checksum() 420 ip_checksum.c 1578
ip_header() 420 ip_header.c 1580 ip_mask() 420
ip_mask.c 1581 ip_public.c 1582 ip_reference() 420
ip_reference.c 1582 ip_rx() 420 ip_rx.c 1583
ip_table[] 422 ip_tx() 420 ip_tx.c 1590 IRET 53
irq_off() 275 irq_off.c 1301 irq_on() 275 irq_on.c
1302 isatty() 683 isatty.c 2211 isr.s 287 1303
isr_exception_name() 284 isr_exception_name.c
1327 isr_exception_unrecoverable()
284 isr_exception_unrecoverable.c 1328
isr_irq_clear() 284 isr_irq_clear.c 1329
isr_irq_clear_pic1() 284 isr_irq_clear_pic1.c
1330 isr_irq_clear_pic2() 284
isr_irq_clear_pic2.c 1331 isr_n() 284 issue 771
I&D 243 kbd.h 899 1071 kbd_isr() 357 kbd_isr.c 1072
kbd_load() 357 kbd_load.c 1077 kbd_public.c 1081
kbd_t 252 kernel.ld 262 994 kill 476 kill() 533
kill.c 1895 2343 kmain.c 1508 kmem_arp 757
kmem_file 758 kmem_inode 759 kmem_mmp 759 kmem_net
760 kmem_ps 761 kmem_route 762 kmem_sb 763 k_exit.s
1333 k_gets.c 1333 k_perror.c 1334 k_printf.c 1335
k_sleep.c 1336 k_stime.c 1337 k_usleep.c 1337
k_vprintf.c 1339 k_vsprintf.c 1340 labs() 604
labs.c 2029 ldiv() 620 ldiv.c 2030 LGDT 42 libgen.h
1865 lib_k.h 900 1332 lib_s.h 901 1341 LIDT 50
limits.h 1802 link() 534 link.c 2213 listen() 537
listen.c 2136 llabs() 604 llabs.c 2030 lldiv() 620

lldiv.c 2031 ln 478 ln.c 2349 localtime() 617 login
479 login.c 2353 longjmp() 318 567 longjmp.c 1889 ls
480 ls.c 2358 lseek() 539 lseek.c 2213 main() 267
main.h 902 1505 major() 684 major.c 2155 MAKEDEV 782
makedev() 684 makedev.c 2156 MAKEDEV.c 2233
makeit.sep 995 malloc() 685 malloc.c 2056 man 482
man.c 2369 mboot_cmdline_opt() 261
mboot_cmdline_opt.c 1540 mboot_public.c 1543
mboot_save() 261 mboot_save.c 1543 mb_alloc() 335
mb_alloc.c 1525 mb_alloc_size() 335
mb_alloc_size.c 1528 mb_clean() 335 mb_clean.c
1531 mb_free() 335 mb_free.c 1531 mb_print() 335
mb_print.c 1534 mb_public.c 1536 mb_reduce() 335
mb_reduce.c 1536 mb_reference() 335
mb_reference.c 1538 mb_size() 335 mb_size.c 1538
mem 763 memcpy() 687 memcpy.c 2069 memchr() 688
memchr.c 2070 memcmp() 689 memcmp.c 2070 memcpy()
690 memcpy.c 2071 memmove() 691 memmove.c 2071
memory.c 333 memory.h 333 902 1523 memset() 691
memset.c 2073 MEM_BLOCK_SIZE 333 MEM_MAX_BLOCKS
333 menu.c 1521 minor() 684 minor.c 2156 mkdir 483
mkdir() 540 mkdir.c 2151 2378 mknod() 543 mknod.c
2152 mktime() 617 mktime.c 2172 mmcheck 484
mmcheck.c 2383 more 485 more.c 2390 mount 783
mount() 545 mount.c 2117 2397 multiboot.h 905 1539
multiboot_t 252 *multiboot specification* 256 namep() 693
namep.c 2118 nc 486 nc.c 2399 NE2000 132 ne2k.h 907
1081 ne2k_check() 410 ne2k_check.c 1085 ne2k_isr()
410 ne2k_isr.c 1088 ne2k_isr_expect() 410

ne2k_isr_expect.c 1090 ne2k_reset() 410
ne2k_reset.c 1092 ne2k_rx() 410 ne2k_rx.c 1104
ne2k_rx_reset() 410 ne2k_rx_reset.c 1113
ne2k_tx() 410 ne2k_tx.c 1115 net.h 409 909 1544
net_buffer_eth() 415 net_buffer_eth.c 1594
net_buffer_lo() 415 net_buffer_lo.c 1595
net_eth_ip_tx() 415 net_eth_ip_tx.c 1597
net_eth_tx() 415 net_eth_tx.c 1601 net_index()
415 net_index.c 1602 net_index_eth() 415
net_index_eth.c 1602 net_init() 415 net_init.c
1605 net_print.c 1611 net_public.c 1612 net_rx()
415 net_rx.c 1612 NIC 132 ntohl() 613 ntohl.c 1824
ntohs() 613 ntohs.c 1825 null 764 NULL.h 1799
offsetof() 694 open() 547 open.c 1852 opendir() 695
opendir.c 1829 os32 235 os32.h 339 2091 OUTB 22
out_16() 274 out_8() 274 part.h 912 1685 passwd 771
PATA 79 path_device() 401 882 path_device.c 1234
path_fix() 400 884 path_fix.c 1235 path_full() 400
885 path_full.c 1237 path_inode() 401 886
path_inode.c 1239 path_inode_link() 401 888
path_inode_link.c 1245 PCI 112 pci.h 912 1119
pci_init.c 1122 pci_public.c 1125 *peripheral component
interconnect* 112 perror() 697 perror.c 1921 PIC 66 ping
785 ping.c 2409 pipe 392 pipe() 552 pipe.c 2214 PIT 73
port 765 printf() 698 printf.c 1922 proc.h 286 913
1686 proc_available() 913 proc_available.c 1692
proc_dump_memory() 914 proc_dump_memory.c 1693
proc_init() 306 915 proc_init.c 1695 proc_print()
918 proc_print.c 1701 proc_public.c 1705

proc_reference() 918 proc_reference.c 1705
proc_scheduler() 309 924 proc_scheduler.c 1722
proc_sch_net() 920 proc_sch_net.c 1706
proc_sch_signals() 921 proc_sch_signals.c 1709
proc_sch_terminals() 922 proc_sch_terminals.c
1710 proc_sch_timers() 923 proc_sch_timers.c 1721
proc_sig_chld() 927 proc_sig_chld.c 1728
proc_sig_cont() 928 proc_sig_cont.c 1730
proc_sig_core() 930 proc_sig_core.c 1731
proc_sig_handler() 312 932 proc_sig_handler.c
1733 proc_sig_ignore() 933 proc_sig_ignore.c 1740
proc_sig_off() 934 proc_sig_off.c 1740
proc_sig_on() 934 proc_sig_on.c 1741
proc_sig_status() 936 proc_sig_status.c 1741
proc_sig_stop() 937 proc_sig_stop.c 1742
proc_sig_term() 938 proc_sig_term.c 1742
proc_sys_exec() 940 proc_sys_exec.c 1744 proc_t
252 298 proc_timer_init() 944 proc_timer_init.c
1767 proc_wakeup_pipe_read()
945 proc_wakeup_pipe_read.c
1768 proc_wakeup_pipe_write()
945 proc_wakeup_pipe_write.c
1769 proc_wakeup_terminal() 945
proc_wakeup_terminal.c 1769 *programmable interrupt
controller* 66 *programmable interval timer* 73 ps 487 ps.c 2414
PS/2 77 ptr() 947 ptr.c 1771 ptrdiff_t.h 1804 putc()
651 putchar() 651 putchar.c 1922 putenv() 705
putenv.c 2031 puts() 652 puts.c 1923 pwd.h 1883
pwent.c 1884 qemu 1006 qsort() 707 qsort.c 2034

rand() 709 rand.c 2038 read() 553 read.c 2215
 readdir() 710 readdir.c 1832 realloc() 685
 realloc.c 2063 recvfrom() 555 recvfrom.c 2137
 restrict.h 1804 rewind() 711 rewind.c 1923
 rewinddir() 712 rewinddir.c 1834 rm 489 rm.c 2422
 rmdir 490 rmdir() 557 rmdir.c 2218 2424 route 786
 route.c 2426 route.h 950 1616 routeadd() 559
 routeadd.c 2122 routedel() 560 routedel.c 2124
 route_init() 426 route_init.c 1617 route_print.c
 1618 route_public.c 1619 route_remote_to_local()
 426 route_remote_to_local.c
 1619 route_remote_to_router() 426
 route_remote_to_router.c 1621 route_sort() 426
 route_sort.c 1622 RPL 16 run.c 1522 sa_family_t.h
 2128 sbrk() 503 sbrk.c 2219 sb_inode_status() 373
 867 sb_inode_status.c 1253 sb_mount() 373 869
 sb_mount.c 1255 sb_print() 373 872 sb_print.c 1260
 sb_reference() 373 872 sb_reference.c 1261
 sb_save() 373 874 sb_save.c 1263 sb_t 252 370
 sb_zone_status() 373 867 sb_zone_status.c 1265
 scanf() 713 scanf.c 1924 screen.h 952 1125
 screen_cell() 359 screen_clear() 359
 screen_clear.c 1127 screen_current() 359
 screen_current.c 1128 screen_init() 359
 screen_init.c 1128 screen_newline() 359
 screen_new_line.c 1129 screen_number() 359
 screen_number.c 1130 screen_pointer() 359
 screen_pointer.c 1132 screen_public.c 1132
 screen_putc() 359 screen_putc.c 1133

screen_scroll() 359 screen_scroll.c 1134
 screen_select() 359 screen_select.c 1136
 screen_t 252 screen_update() 359 screen_update.c
 1137 SEEK.h 1800 *selector* 19 selettore 19 send() 562 send.c
 2140 setbuf() 722 setbuf.c 1924 setegid() 564
 setegid.c 2220 setenv() 723 setenv.c 2039
 seteuid() 572 seteuid.c 2220 setgid() 564 setgid.c
 2221 setgrent() 661 setjmp() 318 567 setjmp.h 1887
 setjmp.s 1890 setpgrp() 570 setpgrp.c 2222
 setpwent() 672 setuid() 572 setuid.c 2222
 setvbuf() 722 setvbuf.c 1924 shell 491 shell.c 2429
 signal() 574 signal.c 1896 signal.h 1891 size_t.h
 1805 sleep() 576 sleep.c 2223 snprintf() 698
 snprintf.c 1925 socket 775 socket() 577 socket.c
 2142 socket.h 2129 socklen_t.h 2143
 sock_free_port() 875 sock_free_port.c 1266
 sock_reference() 876 sock_reference.c 1267
specifche *multiboot* 256 sprintf() 698 sprintf.c 1925
 srand() 709 sscanf() 713 sscanf.c 1926 stack.s 262
 1523 stat() 580 stat.c 2152 stat.h 2144 stdarg.h 1805
 stdbool.h 1806 stddef.h 1806 stdint.h 1807 stdio.h
 726 1897 stdlib.h 2013 STI 24 sti() 275 sti.s 1331
 stime() 593 stime.c 2176 strcat() 728 strcat.c 2073
 strchr() 730 strchr.c 2074 strcmp() 731 strcmp.c
 2074 strcoll() 731 strcoll.c 2075 strcpy() 732
 strcpy.c 2075 strcspn() 738 strcspn.c 2076
 strdup() 733 strdup.c 2077 strerror() 734
 strerror.c 2077 string.h 2067 strlen() 735
 strlen.c 2081 strncat() 728 strncat.c 2082

[strncmp\(\)](#) [731](#) [strncmp.c](#) [2082](#) [strncpy\(\)](#) [732](#)
[strncpy.c](#) [2083](#) [strpbrk\(\)](#) [736](#) [strpbrk.c](#) [2084](#)
[strrchr\(\)](#) [730](#) [strrchr.c](#) [2084](#) [strspn\(\)](#) [738](#) [strspn.c](#)
[2085](#) [strstr\(\)](#) [739](#) [strstr.c](#) [2086](#) [strtok\(\)](#) [739](#)
[strtok.c](#) [2087](#) [strtol\(\)](#) [743](#) [strtol.c](#) [2043](#) [strtoul\(\)](#)
[743](#) [strtoul.c](#) [2049](#) [strxfrm\(\)](#) [746](#) [strxfrm.c](#) [2091](#)
[sys\(\)](#) [586](#) [sys.s](#) [2125](#) [syslinux](#) [1007](#) [sysroutine\(\)](#) [305](#)
[307](#) [948](#) [sysroutine.c](#) [1771](#) [s_accept.c](#) [1351](#) [s_bind.c](#)
[1355](#) [s_brk\(\)](#) [901](#) [s_brk.c](#) [1359](#) [s_chdir\(\)](#) [403](#) [901](#)
[s_chdir.c](#) [1367](#) [s_chmod\(\)](#) [403](#) [901](#) [s_chmod.c](#) [1369](#)
[s_chown\(\)](#) [403](#) [406](#) [901](#) [s_chown.c](#) [1370](#) [s_clock\(\)](#) [901](#)
[s_clock.c](#) [1372](#) [s_close\(\)](#) [901](#) [s_close.c](#) [1372](#)
[s_connect.c](#) [1375](#) [s_dup\(\)](#) [406](#) [901](#) [s_dup.c](#) [1381](#)
[s_dup2\(\)](#) [406](#) [901](#) [s_dup2.c](#) [1381](#) [s_fchmod\(\)](#) [406](#) [901](#)
[s_fchmod.c](#) [1383](#) [s_fchown\(\)](#) [901](#) [s_fchown.c](#) [1384](#)
[s_fcntl\(\)](#) [406](#) [901](#) [s_fcntl.c](#) [1386](#) [s_fork\(\)](#) [901](#)
[s_fork.c](#) [1388](#) [s_fstat\(\)](#) [406](#) [901](#) [s_fstat.c](#) [1398](#)
[s_ipconfig.c](#) [1400](#) [s_kill\(\)](#) [901](#) [s_kill.c](#) [1402](#)
[s_link\(\)](#) [403](#) [901](#) [s_link.c](#) [1406](#) [s_listen.c](#) [1408](#)
[s_longjmp\(\)](#) [318](#) [901](#) [s_longjmp.c](#) [1410](#) [s_lseek\(\)](#) [406](#)
[901](#) [s_lseek.c](#) [1412](#) [s_mkdir\(\)](#) [403](#) [901](#) [s_mkdir.c](#) [1414](#)
[s_mknod\(\)](#) [403](#) [901](#) [s_mknod.c](#) [1418](#) [s_mount\(\)](#) [403](#) [901](#)
[s_mount.c](#) [1421](#) [s_open\(\)](#) [403](#) [901](#) [s_open.c](#) [1423](#)
[s_pipe\(\)](#) [406](#) [901](#) [s_pipe.c](#) [1432](#) [s_read\(\)](#) [406](#) [901](#)
[s_read.c](#) [1435](#) [s_recvfrom.c](#) [1441](#) [s_routeadd.c](#) [1456](#)
[s_routedel.c](#) [1458](#) [s_sbrk\(\)](#) [901](#) [s_sbrk.c](#) [1460](#)
[s_send.c](#) [1462](#) [s_setegid\(\)](#) [901](#) [s_setegid.c](#) [1469](#)
[s_seteuid\(\)](#) [901](#) [s_seteuid.c](#) [1470](#) [s_setgid\(\)](#) [901](#)
[s_setgid.c](#) [1471](#) [s_setjmp\(\)](#) [318](#) [901](#) [s_setjmp.c](#) [1472](#)

s_setuid() 901 s_setuid.c 1474 s_signal() 901
s_signal.c 1475 s_socket.c 1477 s_stat() 403 901
s_stat.c 1480 s_stime() 901 s_stime.c 1483
s_tcgetattr() 901 s_tcgetattr.c 1484
s_tcsetattr() 901 s_tcsetattr.c 1486 s_time() 901
s_time.c 1488 s_umount() 403 901 s_umount.c 1489
s_unlink() 403 901 s_unlink.c 1493 s_wait() 901
s_wait.c 1498 s_write() 406 901 s_write.c 1500
s__exit() 901 s__exit.c 1346 tap0 1008 tastiera 77
tcgetattr() 588 tcgetattr.c 2161 TCP 211 tcp() 433
tcp.c 1628 tcp.h 955 1627 1879 tcp_close() 433
tcp_close.c 1653 tcp_connect() 433 tcp_connect.c
1656 tcp_rx_ack() 433 tcp_rx_ack.c 1658
tcp_rx_data() 433 tcp_rx_data.c 1662 tcp_show() 433
tcp_show.c 1664 tcp_status.c 1666 tcp_test.c 1668
tcp_tx_ack() 433 tcp_tx_ack.c 1669
tcp_tx_raw() 433 tcp_tx_raw.c 1671 tcp_tx_rst() 433
tcp_tx_rst.c 1674 tcp_tx_sock() 433
tcp_tx_sock.c 1677 tcsetattr() 588 tcsetattr.c 2161
termios.h 2158 time() 593 time.c 2177 time.h 2162
time_t.h 1810 touch 493 touch.c 2473 tty 493 766
tty.c 2475 tty.h 959 1139 ttyname() 747 ttyname.c 2224
tty_console() 353 tty_console.c 1140
tty_init() 353 tty_init.c 1141 tty_public.c 1143
tty_read() 353 tty_read.c 1144 tty_reference() 353
tty_reference.c 1145 tty_t 252 tty_write() 353
tty_write.c 1146 types.h 2154 t_fcntl.c 2436
t_fifo.c 2437 t_grp.c 2440 t_nc.c 2441 t_ping2.c 2451
t_pipe.c 2453 t_read.c 2456 t_ret.c 2458

t_rx_udp.c 2458 t_scr.c 2461 t_setjmp.c 2462
t_sig.c 2463 t_sig2.c 2464 t_tx_tcp.c 2466
t_tx_udp.c 2469 UDP 205 udp.h 1682 1882 udp_tx() 433
udp_tx.c 1683 umask() 594 umask.c 2154 umount 783
umount() 545 umount.c 2125 2477 unistd.h 2177
unlink() 596 unlink.c 2226 unsetenv() 723
unsetenv.c 2049 utime.c 2229 utime.h 2228 u-area 298
vfprintf() 748 vfprintf.c 1926 vfscanf() 751
vfscanf.c 1927 vfscanf.c 1928 VGA 25 vprintf() 748
vprintf.c 1973 vscanf() 751 vscanf.c 1974
vsnprintf() 748 vsnprintf.c 1975 vsprintf() 748
vsprintf.c 2012 vsscanf() 751 vsscanf.c 2013
wait() 597 wait.c 2157 wait.h 2156 wchar_t.h 1810
write() 598 write.c 2226 yes 494 yes.c 2478 zero 766
zno_t 252 zone_alloc() 376 878 zone_alloc.c 1268
zone_free() 376 878 zone_free.c 1271 zone_print()
376 880 zone_print.c 1273 zone_read() 376 880
zone_read.c 1274 zone_write() 376 880
zone_write.c 1275 z_perror() 600 z_perror.c 2126
z_printf() 600 z_printf.c 2127 z_vprintf() 600
z_vprintf.c 2128 _alloc_list.c 2052 _Exit() 497
_exit() 497 _exit.c 2182 _Exit.c 2017 _gcc.h 1811
_in_16() 274 _in_16.s 1283 _in_32.s 1284 _in_8()
274 _in_8.s 1285 _lldiv.c 1813 _out_16() 274
_out_16.s 1286 _out_32.s 1287 _out_8() 274
_out_8.s 1288 _sighandler_wrapper.s 1893
__udivdi3.c 1815 __divdi3.c 1812 __moddi3.c 1812
__udivdi3.c 1813 __umoddi3.c 1813