Capitolo 68

Introduzione alle estensioni POSIX

68.1 Dal C a POSIX 912
68.1.1 Byte di otto bit 913
68.1.2 Errori 914
68.1.3 Gestione dei file 915
68.1.4 Altre funzionalità particolari 915
68.2 Espressioni regolari POSIX 916
68.2.1 Compilazione, confronto e rilascio della memoria 918
68.2.2 Compilazione dell'espressione regolare 919
68.2.3 Liberazione della memoria 923
68.2.4 Confronto 923
68.2.5 Estrapolazione di sottostringhe 927
68.2.6 Informazioni diagnostiche 932
68.2.7 Esempio completo 936
68.3 Avvio e conclusione dei processi 940
68.3.1 Biforcazione: «fork» 940
68.3.2 Sostituzione: «exec» 943
68.3.3 Attesa della conclusione di un processo figlio 945
68.3.4 Adozione dei processi 946
68.3.5 Gli zombie 948
68.3.6 Variabili di ambiente 950
68.3.7 Variabili di ambiente e avvio di un programma 951

«a2» 2013.11.11 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://informaticalibera.net

68.4 Nozioni sui thread POSIX 953

68.4.1 Numero identificativo del thread 954

68.4.2 Creazione e conclusione di un thread aggiuntivo . . 955

68.4.3 Caratteristiche della funzione che costituisce un thread aggiuntivo 956

68.4.4 Avvio di thread separati e fusione successiva 957

68.4.5 Conflitto nell'accesso ai dati 962

68.4.6 Accesso alle risorse in modo mutualmente esclusivo 966

68.4.7 Accesso esclusivo, ma condizionato 972

68.4.8 Osservazioni finali 978

68.5 I file secondo i sistemi POSIX 979

68.5.1 Apertura e chiusura di un file 980

68.5.2 Lettura e scrittura 987

68.5.3 Spostamento dell'indicatore interno al file 992

68.5.4 Controllo degli errori 995

68.6 Il file system Unix e la sua gestione tipica 996

68.6.1 Il blocco 996

68.6.2 Il super blocco 997

68.6.3 Gli inode 998

68.6.4 La directory 1001

68.6.5 Tabelle del sistema operativo 1002

68.7 Il file system Minix 1 1008

68.7.1 Blocchi e zone 1009
68.7.2 Struttura generale 1010
68.7.3 Super blocco 1011
68.7.4 Mappa di inode 1013
68.7.5 Mappa di zone 1015
68.7.6 Inode 1016
68.7.7 Directory 1020
68.7.8 Il problema dell'inversione dei byte 1022

68.8 Creazione ed eliminazione di file di qualunque tipo . . 1022

68.8.1 La funzione «mknod()» 1023
68.8.2 La funzione «mkdir()» 1031
68.8.3 La funzione «mkfifo()» 1032
68.8.4 La funzione «unlink()» 1032
68.8.5 La funzione «rmdir()» 1033
68.8.6 La funzione «remove()» 1034

68.9 Condotti 1034

68.9.1 Realizzazione materiale del condotto 1035
68.9.2 Condotti «senza nome» e condotti «con nome» . . 1037
68.9.3 Protocollo di accesso ai condotti 1037
68.9.4 Funzione «pipe()» 1038
68.9.5 Esempio di condotto attraverso un file FIFO 1043

68.10 Lettura delle directory 1047

68.10.1 Tipi derivati 1048
68.10.2 Procedura per accedere a una directory 1048

68.10.3 Attributo «FD_CLOEXEC» 1049
68.10.4 Esempio di utilizzo delle funzioni di accesso alle directory 1050

68.11 Riferimenti 1051

close() 980 closedir() 1048 creat() 980 DIR 1048 dirent.h 1047 environ 950 errno 995 execl() 943 execve() 951 fcntl.h 979 FD_CLOEXEC 1049 fork() 940 init 946 lseek() 992 major() 1028 makedev() 1028 minor() 1028 mkdir() 1031 mkfifo() 1032 1043 mknod() 1023 open() 980 opendir() 1048 O_APPEND 980 O_CREAT 980 O_EXCL 980 O_NOCTTY 980 O_NONBLOCK 980 O_RDONLY 980 O_RDWR 980 O_SYNC 980 O_TRUNC 980 O_WRONLY 980 pipe() 1038 pthread_t 954 read() 987 readdir() 1048 regcomp() 918 919 regerror() 918 932 regex.h 918 regexec() 918 923 927 regex_t 918 919 regfree() 918 923 regmatch_t 918 923 927 remove() 1034 rewinddir() 1048 rmdir() 1033 stat.h 979 struct dirent 1048 S_IRGRP 980 S_IROTH 980 S_IRUSR 980 S_IRWXG 980 S_IRWXO 980 S_IRWXU 980 S_ISGID 980 S_ISUID 980 S_ISVTX 980 S_IWGRP 980 S_IWOTH 980 S_IWUSR 980 S_IXGRP 980 S_IXOTH 980 S_IXUSR 980 unistd.h 979 unlink() 1032 wait() 945 write() 987

68.1 Dal C a POSIX

Lo standard del linguaggio C è definito in modo tale da consentirne l'uso in contesti architetturali molto diversi tra loro. Tuttavia, il linguaggio C è nato per i sistemi Unix e quando si vuole considera-

re una libreria di funzioni più ampia, rispetto allo standard del linguaggio, ci si riferisce normalmente allo standard POSIX. Pertanto, lo standard POSIX estende la libreria del linguaggio C, con funzionalità che dipendono da un'organizzazione del sistema operativo conforme, almeno fondamentalmente, a quella di Unix.

Logicamente, i sistemi operativi possono essere più o meno conformi con lo standard POSIX e l'utilizzo delle estensioni introdotte da questo standard va preceduto da una ricerca sulla loro compatibilità.

In generale va osservato che dal C puro a POSIX, la filosofia di programmazione cambia leggermente e occorre considerare alcuni particolari.

68.1.1 Byte di otto bit

«

Lo standard C prescrive che un byte sia di almeno 8 bit, mentre secondo POSIX, il byte diventa necessariamente di 8 bit. Infatti, nel file di intestazione '`stdint.h`', secondo lo standard C la definizione dei tipi derivati '**[u]int*n*_t**' è facoltativa, ma nello standard POSIX diviene obbligatoria. Così facendo, dato che diviene obbligatorio disporre dei tipi derivati '**int8_t**' e '**uint8_t**', non avendo un altro modo per definirli, è necessario che il tipo '**char**' sia esattamente di 8 bit:

```
typedef signed char              int8_t;
…
typedef unsigned char            uint8_t;
```

Per lo stesso motivo, tutti gli altri tipi interi devono essere multipli (al quadrato), del byte, ma oltre a questo, quando la CPU fosse anche in grado di gestire facilmente interi più grandi di 32 bit, il tipo '**int**' potrebbe essere al massimo di soli 32 bit. Per esempio, dispo-

nendo di una CPU a 128 bit, si è praticamente costretti a dichiarare questi tipi derivati nel modo seguente, riservando, evidentemente, il tipo '**`long long int`**' per gli interi a 128 bit che per l'architettura sarebbero invece normali:

```
typedef signed char                 int8_t;
typedef short int                   int16_t;
typedef int                         int32_t;
typedef long int                    int64_t;
//typedef long long int             int128_t;    // non standard

typedef unsigned char               uint8_t;
typedef unsigned short int          uint16_t;
typedef unsigned int                uint32_t;
typedef unsigned long int           uint64_t;
//typedef unsigned long long int uint128_t;   // non standard
```

68.1.2 Errori

«

Molte funzioni, aggiunte dallo standard POSIX, che restituiscono un valore intero, utilizzano il valore -1 per dichiarare la presenza di un errore che ha prodotto un esito non valido. In tal caso, si possono avere funzioni che restituiscono un valore, che, se maggiore o uguale a zero, è valido, mentre se è negativo non lo è, oppure funzioni che semplicemente restituiscono zero o -1.

Questo fatto si scontra con il principio per cui lo zero equivale a *Falso* e un valore diverso da zero equivale a *Vero*, perché in questo caso occorre verificare precisamente il valore, oppure occorre invertirlo logicamente perché abbia il significato atteso.

Questa inversione logica dei risultati si riscontra anche nei programmi di servizio di un sistema operativo POSIX, per cui il linguaggio

della shell POSIX considera lo zero come un valore pari a *Vero* e qualunque altro valore pari a *Falso*.

Oltre a questo fatto, i tipi di errore che si possono annotare nella variabile ***errno*** sono molto più numerosi rispetto a quanto previsto dallo standard C, pertanto aumentano le macro-variabili previste nel file '`errno.h`'.

68.1.3 Gestione dei file

La libreria standard del linguaggio C prevede una gestione dei file attraverso dei «flussi», mentre la libreria POSIX introduce il concetto di descrittore del file (*file descriptor*), ovvero un semplice numero intero.[1]

In uno stesso programma si possono usare entrambe le modalità di gestione dei file, ma è evidente che va evitato l'uso simultaneo sullo stesso file.

A parte la differenza nel modo di identificare un file aperto in un programma, la libreria POSIX mette in condizione di accedere alla gestione dei permessi e delle altre caratteristiche dei file, secondo le convenzioni di un sistema Unix; inoltre offre una gestione ordinata dei blocchi di accesso a porzioni degli stessi (*lock*).

68.1.4 Altre funzionalità particolari

La libreria POSIX offre anche altre funzionalità che possono essere importanti. Per esempio definisce delle funzioni per il trattamento delle espressioni regolari e per la gestione dei thread multipli.

68.2 Espressioni regolari POSIX

«

Un’espressione regolare è un modello che descrive la corrispondenza con una porzione di una stringa. Le espressioni regolari sono costruite, in maniera analoga alle espressioni matematiche, combinando espressioni più brevi. Lo standard POSIX distingue due tipi di espressioni regolari: quelle elementari, o BRE, e quelle estese, o ERE (si vedano in particolare le sezioni dedicate alle espressioni regolari in generale: 23.1 e 23.2). Lo standard POSIX prevede una libreria specifica per la gestione delle espressioni regolari, a cui si fa riferimento in questo capitolo.

Va osservato che nel caso del compilatore GCC, con le librerie GNU, per la compilazione sono sufficienti le librerie principali, le quali vengono incluse automaticamente.

Tabella 68.3. Schema sintetico delle espressioni regolari, secondo la definizione POSIX: operatori fondamentali.

<table>
<tr><th>Descrizione</th><th>BRE POSIX</th><th>ERE POSIX</th></tr>
<tr><td>Escape: protezione del carattere successivo o attribuzione a tale carattere di un significato speciale.</td><td>\</td><td>\</td></tr>
<tr><td>Ancora all’inizio della riga o della stringa.</td><td>^</td><td>^</td></tr>
<tr><td>Ancora alla fine della riga o della stringa.</td><td>$</td><td>$</td></tr>
<tr><td>Alternativa.</td><td></td><td>|</td></tr>
<tr><td>Raggruppamento.</td><td>\(\)</td><td>( )</td></tr>
<tr><td>Elenco (individua un solo carattere).</td><td>[ ]</td><td>[ ]</td></tr>
</table>

Descrizione	BRE POSIX	ERE POSIX
Riferimento al raggruppamento ***n*** esimo.	`\`***n***	

Tabella 68.4. Schema sintetico delle espressioni regolari, secondo la definizione POSIX: operatori interni alle espressioni tra parentesi quadre.

Descrizione	BRE POSIX	ERE POSIX
Sequenze.	***xy***...	***xy***...
Intervalli.	***x***-***y***	***x***-***y***
Elementi di collazione.	`[.    .]`	`[.    .]`
Caratteri equivalenti.	`[=    =]`	`[=    =]`
Classi di caratteri.	`[:    :]`	`[:    :]`

Tabella 68.5. Schema sintetico delle espressioni regolari, secondo la definizione POSIX: operatori di ripetizione.

Descrizione	BRE POSIX	ERE POSIX
Zero o più ripetizioni del carattere ***x***.	***x***`*`	***x***`*`
Una o nessuna occorrenza del carattere ***x***.		***x***`?`
Una o più ripetizioni del carattere ***x***.		***x***`+`
Esattamente ***n*** volte il carattere ***x***.	***x***`\{`***n***`\}`	***x***`{`***n***`}`
Almeno ***n*** volte ***x***.	***x***`\{`***n***`,\}`	***x***`{`***n***`,}`

Descrizione	BRE POSIX	ERE POSIX
Da ***n*** a ***m*** volte ***x***.	`x\{n,m\}`	`x{n,m}`

68.2.1 Compilazione, confronto e rilascio della memoria

«

Per eseguire un confronto con un'espressione regolare, attraverso le funzioni definite dallo standard POSIX, è necessario prima tradurre l'espressione regolare in una variabile strutturata di tipo '**`regex_t`**'. Tale operazione di analisi e traduzione dell'espressione regolare viene definita dallo standard come «compilazione». La compilazione dell'espressione regolare avviene con la funzione ***regcomp()***, con la quale, oltre che fornire la stringa contenente l'espressione regolare stessa, si devono specificare delle opzioni sul modo in cui interpretarla o gestirla. Per esempio, in questa fase va stabilito se l'espressione è di tipo BRE o ERE, se conta la differenza tra lettere maiuscole e minuscole, se si intendono estrapolare delle sottostringhe attraverso la comparazione, se il codice di interruzione di riga ha un qualche valore particolare o meno.

Quando si dispone di un'espressione regolare compilata, si può passare alla comparazione di questa con una stringa, attraverso la funzione ***regexec()***, con la quale si possono dare delle opzioni aggiuntive, nel modo finale di effettuare il confronto. Per la comparazione può essere necessaria la definizione di una variabile strutturata, di tipo '**`regmatch_t`**'.

Quando la variabile strutturata contenente l'espressione regolare non serve più, va rilasciata espressamente la memoria a cui gli elementi della stessa fanno riferimento. In altri termini, non basta liberare

la memoria della variabile che rappresenta la struttura, perché rimarrebbero allocati altri dati raggiunti attraverso dei puntatori. Per liberare un'espressione regolare compilata si utilizza la funzione ***regfree()***.

Le funzioni ***regcomp()*** e ***regexec()***, hanno la caratteristica di restituire un valore intero, pari a zero se l'operazione è stata conclusa con successo, oppure un valore diverso se si è verificato qualche tipo di problema. I valori restituiti, se diversi da zero, sono codificati ordinatamente da macro-variabili simboliche appropriate. Eventualmente, con la funzione ***regerror()***, è possibile ottenere la traduzione dell'errore, associato al riferimento all'espressione regolare compilata, in una stringa più esplicita.

Per utilizzare le espressioni regolari POSIX è necessario includere inizialmente il file di intestazione '`regex.h`'.

68.2.2 Compilazione dell'espressione regolare

«

Un'espressione regolare, in forma di stringa, viene compilata attraverso la funzione ***regcomp()***, inserendo i dati necessari in una struttura di tipo '**`regex_t`**'.

```
int regcomp (regex_t *restrict re, const char *restrict regex,
             int cflags);
```

Il prototipo della funzione mostra che il primo parametro, ***re***, deve essere un puntatore al tipo '**`regex_t`**': si tratta della struttura che viene modificata attraverso la compilazione. Il secondo parametro, ***regex***, è la stringa che descrive l'espressione regolare (la stringa deve essere terminata regolarmente con un carattere nullo, come di

consueto). L'ultimo parametro, '**cflags**', è un numero intero i cui bit descrivono le opzioni da considerare per l'interpretazione corretta della stringa dell'espressione regolare; tali bit vengono composti assieme attraverso l'uso di macro-variabili simboliche che fanno parte della stessa libreria della funzione ***regcomp()***.

Tabella 68.6. Il tipo '**regex_t**' definisce una variabile strutturata che contiene almeno il membro '**re_sub**'.

Tipo	Nome	Descrizione
`size_t`	`re_sub`	Quantità di sottoespressioni tra parentesi tonde.

Va osservato che l'espressione regolare viene fornita attraverso una stringa «normale», ovvero un array di '**char**'.

L'esempio seguente dovrebbe servire a comprendere l'uso della funzione ***regcomp()***:

```
#include <regex.h>
...
    regex_t re;
    regcomp (&re, "01[a-z]*", REG_EXTENDED|REG_NOSUB);
...
```

Come si può osservare, viene dichiarata una variabile di tipo '**regex_t**', della quale viene fornito il puntatore nella chiamata di ***regcomp()***; inoltre, l'ultimo argomento della funzione è composto utilizzando due macro-variabili simboliche, sommate assieme con l'operatore '**|**', ovvero un OR bit per bit.

Tabella 68.8. Macro-variabili simboliche da usare come opzioni per la compilazione di un'espressione regolare.

Macro-variabile	Significato
`REG_EXTENDED`	L'espressione regolare fornita è di tipo ERE (estesa). Se non si usa questa opzione, si intende che l'espressione sia di tipo BRE.
`REG_ICASE`	L'espressione regolare fornita va valutata senza distinguere tra lettere maiuscole o minuscole.
`REG_NOSUB`	Per la compilazione dell'espressione regolare non si intende tenere conto della corrispondenza eventuale di sottostringhe; in altri termini, non si vogliono considerare le parentesi '**\(**' e '**\)**', oppure '**(**' e '**)**' (a seconda che si tratti di ERE o BRE). In tal caso, l'espressione regolare serve per il confronto, ma non per estrapolare porzioni del risultato ottenuto.
`REG_NEWLINE`	In condizioni normali, il codice *new-line* contenuto nella stringa con cui l'espressione regolare deve essere confrontata, viene trattato come gli altri caratteri. Con l'opzione '**REG_NEWLINE**', invece, l'operatore '**^**' individua l'inizio di un testo che segue un codice *new-line*, mentre l'operatore '**$**' individua la fine di un testo che precede un codice *new-line*.

La funzione ***regcomp()*** restituisce zero se il procedimento di compilazione dell'espressione regolare termina regolarmente, senza problemi nell'interpretazione della stringa che la rappresenta; altrimenti

restituisce un valore diverso che rappresenta un errore. Per poter valutare l'errore, occorre fare un confronto con delle macro-variabili simboliche, come descritto nella tabella successiva.

Tabella 68.9. Macro-variabili simboliche che rappresentano il tipo di errore restituito dalla funzione ***regcomp()***.

Macro-variabile	Significato
`REG_BADBR`	Il contenuto di '`\{...\}`' (nel caso di BRE) o di '`{...}`' (nel caso di ERE), risulta non valido: potrebbe non trattarsi di un numero, oppure potrebbe esserci un numero troppo grande, oppure potrebbero esserci più di due numeri, oppure il primo potrebbe essere più grande del secondo. Infatti, il contenuto di tale raggruppamento deve essere un numero singolo, oppure due numeri separati da una virgola, dove il primo deve essere inferiore al secondo.
`REG_BADPAT`	Espressione regolare non valida (errore di sintassi).
`REG_BADRPT`	Un operatore di ripetizione, del tipo '`?`', '`*`' o '`+`', non è preceduto a un'espressione regolare, ovvero si trova in una posizione sbagliata.
`REG_ECOLLATE`	Elemento di collazione (*collating element*) non valido, nell'ambito della configurazione locale attuale.
`REG_ECTYPE`	Riferimento a una classe di caratteri non valida.
`REG_EESCAPE`	L'espressione regolare termina con '`\`' e ciò non è ammissibile.
`REG_ESUBREG`	Una sequenza '***n***', dove ***n*** è un numero, è errata.

Macro-variabile	Significato
`REG_EBRACE`	Le parentesi graffe che descrivono la ripetizione di qualcosa non bilanciano. Può trattarsi di sequenze del tipo '`\{...\}`' per le espressioni BRE o del tipo '`{...}`' per le espressioni ERE.
`REG_EBRACK`	Parentesi quadre non bilanciate (parentesi aperta e non chiusa, o viceversa).
`REG_EPAREN`	Le parentesi tonde che descrivono delle sottoespressioni non bilanciano. Può trattarsi di sequenze del tipo '`\(...\)`' per le espressioni BRE o del tipo '`(...)`' per le espressioni ERE.
`REG_ERANGE`	Un'estremità di un intervallo di valori non è valido.
`REG_ESPACE`	Il procedimento di interpretazione dell'espressione regolare porta all'esaurimento della memoria disponibile.

68.2.3 Liberazione della memoria

«

Un'espressione regolare compilata occupa memoria, non solo nella variabile strutturata che la rappresenta, ma anche in altre aree a cui il contenuto di tale variabile può puntare. Quando l'espressione regolare non serve più, la memoria relativa va liberata esplicitamente, attraverso la funzione ***regfree()***, la quale non restituisce alcunché e richiede di indicare solo il puntatore alla variabile strutturata che rappresenta l'espressione regolare stessa.

```
void regfree (regex_t *re);
```

68.2.4 Confronto

«

La comparazione di un'espressione regolare compilata e di una stringa, si svolge con la funzione ***regexec()***. Questa funzione richiede diversi argomenti, perché ci deve essere la possibilità di estrapolare anche delle sottostringhe, corrispondenti a delle sottoespressioni racchiuse tra parentesi tonde.

```
int regexec (const regex_t *restrict re,
             const char *restrict stringa,
             size_t n_match, regmatch_t p_match[restrict],
             int eflags);
```

Il prototipo della funzione ***regexec()*** può apparire inizialmente complicato da interpretare. I primi due parametri sono sostanzialmente l'espressione regolare compilata e la stringa da confrontare. Il terzo parametro rappresenta la quantità di elementi dell'array che viene fornito come quarto parametro (di tipo '**`regmatch_t`**'). L'ultimo parametro rappresenta delle opzioni da applicare in fase di comparazione.

Per comprendere l'utilizzo della funzione, inizialmente conviene lasciare da parte i parametri ***n_match*** e ***pmatch***. Così facendo è possibile verificare se l'espressione regolare trova una corrispondenza nella stringa fornita. L'esempio seguente (che dovrebbe essere disponibile presso *allegati/c/esempio-posix-regex-1.c*) mostra la dichiarazione di una funzione che svolge tutti i passaggi, dalla compilazione alla liberazione della memoria.

Listato 68.10. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/t4ZYxS51* , *http://ideone.com/oN1I1* .

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

int
regex_match (char *pattern, const char *string)
{
    int        status;
    regex_t    re;
    status = regcomp (&re, pattern, REG_EXTENDED|REG_NOSUB);
    if (status != 0)
      {
        return (0);
      }
    status = regexec (&re, string, (size_t) 0, NULL, 0);
    if (status != 0)
      {
        regfree (&re);
        return (0);
      }
    regfree (&re);
    return (1);
}

int
main (void)
{
    char *string       = "Ciao amore mio";
    char *re           = "iao";

    if (regex_match (re, string))
      {
```

```
        printf ("Il modello \"%s\" trova corrispondenza ",
                re);
        printf ("nella stringa \"%s\"\n", string);
      }
    else
      {
        printf ("Il modello \"%s\" ", re);
        printf ("NON trova corrispondenza ");
        printf ("nella stringa \"%s\"\n", string);
      }
    return 0;
}
```

In pratica, la funzione ***regexec()*** viene usata semplicemente fornendo il puntatore alla variabile strutturata contenente l'espressione regolare compilata e la stringa da confrontare:

```
regexec (&re, string, (size_t) 0, NULL, 0);
```

Le opzioni che possono essere indicate alla funzione ***regexec()***, come ultimo argomento, sono solo due e riguardano la facoltà di considerare l'inizio o la fine della stringa come l'inizio o la fine di una riga.

Tabella 68.12. Macro-variabili simboliche da usare come opzioni per la comparazione di una stringa con un'espressione regolare.

Macro-variabile	Significato
`REG_NOTBOL`	In condizioni normali, il carattere '^' trova corrispondenza con l'inizio di una stringa. Con questa opzione, si inibisce tale corrispondenza (*not begin of line*).

Macro-variabile	Significato
`REG_NOTEOL`	In condizioni normali, il carattere '`$`' trova corrispondenza con la fine di una stringa. Con questa opzione, si inibisce tale corrispondenza (*not end of line*).

Il valore restituito dalla funzione ***regexec()*** è zero se il confronto avviene con successo, diversamente si ha un valore diverso da zero, per indicare la mancanza di corrispondenza o l'utilizzo eccessivo di memoria. Va osservato che la macro-variabile ***REG_ESPACE*** è la stessa già vista per la funzione ***regcomp()*** e che ***REG_NOTBOL*** rappresenta, opportunamente, un valore differente da tutte le altre macro-variabili ***REG_***....

Tabella 68.13. Macro-variabili simboliche che rappresentano il tipo di errore (o di insuccesso) restituito dalla funzione ***regexec()***.

Macro-variabile	Significato
`REG_ESPACE`	Il procedimento di confronto porta all'esaurimento della memoria disponibile.
`REG_NOMATCH`	Non c'è corrispondenza tra espressione regolare e stringa.

68.2.5 Estrapolazione di sottostringhe

«

Quando un'espressione regolare contiene una porzione del proprio codice racchiuso tra '`\(`' e '`\)`', nel caso di espressioni BRE, oppure tra '`(`' e '`)`', nel caso di espressioni ERE, è possibile estrapolare la porzione di stringa che corrisponde a tale sottoespressione. Per fare

questo si usa un array di variabili strutturate di tipo '**`regmatch_t`**' che viene fornito come argomento della chiamata di ***regexec()***.

Della variabile strutturata di tipo '**`regmatch_t`**' si sa solo che contiene almeno due campi, denominati '**`rm_so`**' e '**`rm_eo`**' (*regular expression match*: *start offset* e *end offset*). I due campi in questione, sono, a loro volta, di tipo '**`regoff_t`**', corrispondente a un valore intero con segno, di rango appropriato.

Tabella 68.14. Organizzazione di una variabile strutturata di tipo '**`regmatch_t`**'.

Tipo	Nome del membro	Descrizione
`regoff_t`	`rm_so`	Scostamento in byte, dall'inizio della stringa, corrispondente all'inizio della sottostringa individuata.
`regoff_t`	`rm_eo`	Scostamento in byte, dall'inizio della stringa, corrispondente al carattere **successivo** alla sottostringa individuata.

La funzione ***regexec()*** popola il contenuto dell'array di elementi '**`regmatch_t`**', utilizzando il primo elemento (indice 0) per individuare la sottostringa corrispondente all'espressione regolare nel suo complesso, mentre gli elementi successivi riguardano le sottoespressioni eventuali. Pertanto, le sottoespressioni si trovano a partire dall'indice 1 di tale array.

Perché la funzione ***regexec()*** possa estrapolare delle sottostringhe a partire da sottoespressioni, è necessario che l'espressione regolare sia stata compilata senza l'opzione '**`REG_NOSUB`**'. Infatti, tale opzione viene usata per risparmiare risorse quando si sa che non ci si intende avvalere di tale possibilità.

L'esempio seguente mostra un piccolo programma, completo, in cui la funzione ***regex_match()*** si occupa di verificare la corrispondenza con un'espressione regolare e, se c'è corrispondenza, compila anche un array si stringhe con le sottostringhe estratte. Naturalmente, tale array di stringhe deve essere già stato predisposto prima della chiamata della funzione e deve avere una dimensione adeguata a contenere sia la corrispondenza con l'espressione regolare nel suo complesso, sia la corrispondenza con le altre sottoespressioni eventuali. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-regex-2.c* .

Listato 68.15. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/ymVOGkON* , *http://ideone.com/U9Tka* .

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

int
regex_match (char *pattern, const char *string,
             size_t sub_size, char **sub)
{
    regmatch_t match[sub_size];
    size_t     n;
    size_t     m;
    size_t     d;
```

```
    int         status;
    regex_t     re;
    status = regcomp (&re, pattern, REG_EXTENDED);
    if (status != 0)
      {
        return (status);
      }
    status = regexec (&re, string, sub_size, match, 0);
    if (status == 0)
      {
        for (n=0; n < sub_size; n++)
          {
            for (d = 0, m = match[n].rm_so;
                 m >= 0 && m < match[n].rm_eo;
                 m++, d++)
              {
                sub[n][d] = string[m];
              }
            sub[n][d] = '\0';
          }
      }
    regfree (&re);
    return (status);
}

int
main (void)
{
    int   result;
    char *string       = "Ciao amore mio";
    char *re           = "Ciao (amo)re";
    char  sub0[200];
          sub0[0] = '\0';
    char  sub1[200];
```

```
          sub1[0] = '\0';
     char *sub[] = {sub0, sub1};

     result = regex_match (re, string, 2, sub);

     if (result == 0)
       {
         printf ("Il modello \"%s\" trova corrispondenza ",
                 re);
         printf ("nella stringa \"%s\", precisamente ",
                 string);
         printf ("nella porzione \"%s\", mentre la ",
                 sub[0]);
         printf ("sottostringa estratta è \"%s\".\n",
                 sub[1]);
       }
     else
       {
         printf ("Il modello \"%s\" ", re);
         printf ("NON trova corrispondenza ");
         printf ("nella stringa \"%s\"\n", string);
       }
     return 0;
}
```

Compilando il programma ed eseguendolo con i dati che si vedono, si ottiene la visualizzazione del testo seguente:

```
Il modello "Ciao (amo)re" trova corrispondenza nella stringa
"Ciao amore mio", precisamente nella porzione "Ciao amore",
mentre la sottostringa estratta è "amo".
```

68.2.6 Informazioni diagnostiche

«

Le funzioni ***regcomp()*** e ***regexec()*** restituiscono un valore intero che rappresenta l'esito dell'operazione svolta: se è zero l'operazione ha avuto successo, altrimenti c'è un qualche tipo di problema che può essere individuato confrontando tale valore con una serie di macro-variabili prestabilite. Tuttavia, si può ottenere un risultato tradotto in un testo più comprensibile attraverso la funzione ***regerror()***, la quale richiede l'indicazione del numero dell'errore, del puntatore alla variabile strutturata che rappresenta l'espressione regolare a cui si riferisce il problema e le informazioni necessarie a compilare correttamente una stringa con il messaggio appropriato.

```
size_t regerror (int errore, const regex_t *restrict re,
                 char *restrict testo, size_t dimensione);
```

In pratica, il primo parametro è il numero dell'errore o comunque dell'esito dell'operazione svolta, come restituito dalle funzioni ***regcomp()*** e ***regexec()***; il secondo parametro è il puntatore alla variabile strutturata che rappresenta l'espressione regolare a cui si riferisce l'esito in questione; il terzo parametro è un array di tipo '`char`', in cui la funzione deve poter scrivere il testo della spiegazione; l'ultimo parametro è la dimensione massima di tale array (oltre la quale la funzione non deve scrivere).

Il valore restituito dalla funzione ***regerror()*** è la dimensione utilizzata effettivamente nell'array per scrivere il testo dell'esito (inclusa la terminazione con il byte a zero).

L'esempio seguente mostra un piccolo programma, completo, ottenuto dalla modifica di quello apparso nella sezione precedente, dove

in presenza di un esito non soddisfacente per le funzioni ***regcomp()*** e ***regexec()*** viene visualizzato un messaggio esplicito del problema verificatosi. Il programma richiede volutamente un confronto non corretto per produrre un errore. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-regex-3.c* .

Listato 68.17. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/onqxRRyX* , *http://ideone.com/Q0w9k* .

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

int
regex_match (char *pattern, const char *string,
             size_t sub_size, char **sub)
{
    const int msg_size = 200;
    char      msg[msg_size];
              msg[0] = '\0';
    //
    regmatch_t match[sub_size];
    size_t     n;
    size_t     m;
    size_t     d;
    int        status;
    regex_t    re;
    status = regcomp (&re, pattern, REG_EXTENDED);
    if (status != 0)
      {
        regerror (status, &re, msg, msg_size);
        fprintf (stderr, "%s\n", msg);
        return (status);
      }
    status = regexec (&re, string, sub_size, match, 0);
```

```
    if (status != 0)
      {
        regerror (status, &re, msg, msg_size);
        fprintf (stderr, "%s\n", msg);
      }
    else
      {
        for (n=0; n < sub_size; n++)
          {
            for (d = 0, m = match[n].rm_so;
                 m >= 0 && m < match[n].rm_eo;
                 m++, d++)
              {
                sub[n][d] = string[m];
              }
            sub[n][d] = '\0';
          }
      }
    regfree (&re);
    return (status);
}

int
main (void)
{
    int   result;
    char *string       = "Ciao amore mio";
    char *re           = "*Ciao (amo)re";
    char  sub0[200];
          sub0[0] = '\0';
    char  sub1[200];
          sub1[0] = '\0';
    char *sub[] = {sub0, sub1};
```

```
    result = regex_match (re, string, 2, sub);

    if (result == 0)
      {
        printf ("Il modello \"%s\" trova corrispondenza ",
                re);
        printf ("nella stringa \"%s\", precisamente ",
                string);
        printf ("nella porzione \"%s\", mentre la ",
                sub[0]);
        printf ("sottostringa estratta è \"%s\".\n",
                sub[1]);
      }
    else
      {
        printf ("Il modello \"%s\" ", re);
        printf ("NON trova corrispondenza ");
        printf ("nella stringa \"%s\"\n", string);
      }
    return 0;
}
```

L'espressione regolare contiene un errore che consiste nell'uso dell'asterisco all'inizio della stessa. Provando a eseguire il programma si dovrebbe visualizzare il testo seguente:

```
Invalid preceding regular expression
Il modello "*Ciao (amo)re" NON trova corrispondenza nella stringa
"Ciao amore mio"
```

68.2.7 Esempio completo

«

Viene proposta una funzione per semplificare il confronto di una stringa con un'espressione regolare, più completa rispetto a quanto già mostrato negli esempi delle sezioni precedenti. A sua volta la funzione viene mostrata in un programma di prova completo.

```
int regex_match (char *restrict pattern,
                 const char *restrict string,
                 size_t sub_size,
                 char *restrict sub[restrict],
                 int cflags, int eflags, int verbose);
```

Il primo parametro è la stringa che contiene l'espressione regolare; il secondo parametro è la stringa da confrontare con l'espressione; il terzo parametro è la dimensione massima dell'array che costituisce il quarto parametro; il quarto parametro è un array di puntatori a stringhe, di cui però non si conosce l'ampiezza massima; il quinto parametro è un intero che rappresenta le opzioni da usare con la funzione ***regcomp()***; il sesto parametro è un intero che rappresenta le opzioni da usare con la funzione ***regexec()***; l'ultimo parametro, se diverso da zero, richiede la visualizzazione dei messaggi di errore attraverso lo standard error.

La funzione restituisce un valore pari a zero se tutto il procedimento si completa con successo; altrimenti restituisce l'esito prodotto dalla funzione ***regcomp()*** o da ***regexec()***.

Si osservi che all'inizio del programma è possibile definire la macro-variabile '`restrict`' come commento. Ciò è necessario se il compilatore non riconosce ancora tale parola chiave nella definizione dei

parametri che sono puntatori. Infatti, si tratta di una caratteristica utile solo nei compilatori ottimizzati, in grado di gestire l'elaborazione degli array in modo diverso da quello tradizionale. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-regex-ok.c* .

Listato 68.19. Per provare il codice attraverso un servizio *pastebin*: *http://codepad.org/c06Z8k6Z* , *http://ideone.com/J0O8h* .

```
#include <stdio.h>
#include <regex.h>
//
// #define restrict /**/
//
//
int
regex_match (char *restrict pattern,
             const char *restrict string,
             size_t sub_size, char *restrict sub[restrict],
             int cflags, int eflags, int verbose)
{
    const int  msg_size = 200;
    char       msg[msg_size];
    //
    regmatch_t pmatch[sub_size];
    size_t     n;
    size_t     m;
    size_t     d;
    //
    int        status;
    regex_t    re;
    //
    status = regcomp (&re, pattern, cflags);
    //
    if (status != 0)
```

```
    {
      if (verbose)
        {
          regerror (status, &re, msg, msg_size);
          fprintf (stderr, "%s\n", msg);
        }
      return (status);
    }
  //
  status = regexec (&re, string, sub_size, pmatch,
                    eflags);
  //
  if (status != 0)
    {
      if (verbose)
        {
          regerror (status, &re, msg, msg_size);
          fprintf (stderr, "%s\n", msg);
        }
    }
  else
    {
      for (n=0; n < sub_size; n++)
        {
          for (d = 0, m = pmatch[n].rm_so;
               m >= 0 && m < pmatch[n].rm_eo;
               m++, d++)
            {
              sub[n][d] = string[m];
            }
          sub[n][d] = '\0';
        }
    }
  //
```

```
    regfree (&re);
    //
    return (status);
}
//
//
//
int
main (void)
{
    int   result;
    char *string       = "Ciao amore mio";
    char *re           = "Ciao (amo)re";
    char  sub0[200];
          sub0[0] = '\0';
    char  sub1[200];
          sub1[0] = '\0';
    char *sub[] = {sub0, sub1};
    //
    result = regex_match (re, string, 2, sub, REG_EXTENDED,
                          0, 1);
    //
    if (result == 0)
      {
        printf ("Il modello \"%s\" trova corrispondenza ",
                re);
        printf ("nella stringa \"%s\", precisamente ",
                string);
        printf ("nella porzione \"%s\", mentre la ",
                sub[0]);
        printf ("sottostringa estratta è \"%s\".\n",
                sub[1]);
      }
    else
```

```
        {
          printf ("Il modello \"%s\" ", re);
          printf ("NON trova corrispondenza ");
          printf ("nella stringa \"%s\"\n", string);
        }
    return 0;
}
```

68.3 Avvio e conclusione dei processi

«

In un sistema Unix, l'avvio di un processo si ottiene attraverso l'uso di due chiamate di sistema: una produce una copia del processo esistente, con la possibilità di distinguere poi tra chi è il genitore e chi è invece il figlio; l'altra carica un processo e lo mette in funzione al posto di quello da cui ha avuto origine. Inizialmente, la questione può sembrare complicata o almeno strana. Se però si ha la possibilità di approfondire il funzionamento basilare di un sistema Unix tradizionale, si scopre che è tutto perfettamente logico e lineare, ovvero, che si tratta della scelta progettuale più semplice che si potesse attuare.

Attorno a questi concetti ci sono poi altre questioni legate ai processi, che è bene introdurre assieme al resto, per avere una visione iniziale relativamente completa.

68.3.1 Biforcazione: «fork»

«

Attraverso la funzione ***fork()***, definita nel file di intestazione '`unistd.h`', si ottiene la duplicazione del processo elaborativo corrente, associandogli un numero PID differente, diventando questo figlio del processo da cui la chiamata ha avuto origine. Ciò che va

chiarito è che il processo ottenuto dalla duplicazione continua a funzionare dal punto in cui si trovava il processo originario, pertanto è dal valore restituito dalla funzione ***fork()*** che si riesce a capire se ci si trova a funzionare come genitore o figlio di quel contesto particolare.

Listato 68.20. Per provare il codice attraverso un servizio *pastebin*: http://ideone.com/XpVBY .

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
//----------------------------------------------------------------
int
main (void)
{
    pid_t  pid;
    //
    printf ("Io sono il genitore e il mio numero PID "
            "è %i.\n",
            (int) getpid ());
    pid = fork ();
    if (pid == 0)
      {
        printf ("Io sono il figlio della biforcazione "
                "e il mio genitore ha il numero "
                "PID %i.\n", (int) getppid ());
        exit (0);
      }
    else
      {
        printf ("Ho avviato una biforcazione di me "
                "stesso, la quale ha ottenuto il "
                "numero PID %i.\n", pid);
      }
```

```
    return (0);
}
```

Il listato mostra un esempio completo di programma che avvia una biforcazione di se stesso. La funzione ***fork()*** viene usata senza argomenti e restituisce un numero PID: se questo è diverso da zero, significa che si tratta dell'esecuzione del processo genitore; se invece è zero, è in corso l'esecuzione del processo duplicato. Nell'esempio si fa uso delle funzioni ***getpid()*** e di ***getppid()***, per ottenere, rispettivamente il numero del processo in corso e quello del processo genitore.

Come si può osservare, nell'esempio il processo figlio ha vita breve, perché si limita a dichiarare la propria esistenza, quindi chiama la funzione ***exit()*** per concludere esplicitamente la propria attività.

Il risultato dell'esecuzione di questo programma potrebbe essere costituito dai messaggi seguenti:

```
Io sono il genitore e il mio numero PID è 5531.
Io sono il figlio della biforcazione e il mio genitore ha il ↩
↪numero PID 5531.
Ho avviato una biforcazione di me stesso, la quale ha ↩
↪ottenuto il numero PID 5532.
```

Il processo ottenuto dalla biforcazione è sostanzialmente uguale a quello del genitore (a parte la distinzione del numero PID e della gerarchia genitore-figlio); tuttavia, le differenze emergono in base al livello di complessità del programma in questione. Pertanto, è sempre bene accertarsi nel dettaglio di cosa erediti il processo figlio, dalla pagina di manuale *fork(2)*, quando si vuole usare questa funzione.

Le cose essenziali da sapere riguardano principalmente i file aperti

e i *thread*, ovvero i flussi elaborativi. La biforcazione produce la duplicazione dei file aperti nel nuovo processo, condividendo però l'indice che rappresenta la posizione corrente. In tal modo, le operazioni di lettura e scrittura sui file possono svolgersi in modo coordinato: uno legge fino a un certo punto, l'altro legge da lì fino a un'altra posizione, e lo stesso vale per la scrittura. La biforcazione produce un solo *thread* nel processo figlio, costituito precisamente da quello che ha chiamato la funzione ***fork()***.

68.3.2 Sostituzione: «exec»

«

Un gruppo di funzioni, contraddistinte dal prefisso '`exec`', consente di rimpiazzare il processo corrente con un altro, caricando un programma. Rimpiazzare il processo corrente significa che questo si conclude e, da quel punto, dovrebbe iniziare a funzionare un altro programma dall'inizio.

In condizioni normali, un processo che voglia avviare un programma, esegue prima una biforcazione, quindi, nel codice che riguarda il processo figlio esegue una funzione ***exec...()***, con cui quel figlio viene rimpiazzato con il nuovo programma.

Nell'esempio successivo viene mostrato l'uso della funzione ***execl()***, con la quale si indica il percorso del programma da avviare, seguito dagli argomenti da dare a questo, tenendo conto che il primo deve corrispondere al nome del programma stesso e che l'ultimo deve essere un puntatore a carattere nullo:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
//----------------------------------------------------------
int
main (void)
{
    pid_t  pid;
    int    status;
    //
    pid = fork ();
    if (pid == 0)
      {
        status = execl ("./prog", "prog", (char *) NULL);
        perror (NULL);
        exit (0);
      }
    return (0);
}
```

Se il programma '**`prog`**' viene avviato correttamente dalla directory corrente, come indicato nel percorso di avvio, la funzione ***execl()*** «non ritorna», nel senso che il processo che la avvia scompare. Se invece si verifica un errore, la funzione restituisce il valore -1 e l'esecuzione del processo originario prosegue. In questo caso, si usa la funzione ***perror()*** per visualizzare l'errore annotato nella variabile ***errno***, quindi la funzione ***exit()*** conclude comunque il funzionamento del processo.

68.3.3 Attesa della conclusione di un processo figlio

«

Quando un processo esegue una biforcazione, dalla quale poi si può passare all'esecuzione di un altro programma o meno, ci può essere la necessità di attendere che il processo figlio termini il suo funzionamento. Per fare questo si usa normalmente la funzione ***wait()***, oppure ***waitpid()*** con argomenti appropriati.

Listato 68.23. Per provare il codice attraverso un servizio *pastebin*: http://ideone.com/vD2hs .

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
//----------------------------------------------------------
int
main (void)
{
    pid_t  pid_child;
    pid_t  pid_dead;
    int    status;
    //
    pid_child = fork ();
    if (pid_child == 0)
      {
        sleep (1);
        printf ("ciao!\n");
        exit (7);
      }
    printf ("Ho avviato il processo %i.\n", pid_child);
    //
    pid_dead = wait (&status);
    //
    printf ("Il processo %i si è concluso restituendo "
```

```
                "il valore %x.\n",
                pid_dead, WEXITSTATUS (status));
    //
    return (0);
}
```

L'esempio mostra l'avvio di un processo figlio, in cui, dopo una pausa di un secondo si visualizza un messaggio e quindi quel processo termina restituendo il valore 7. Il processo genitore mostra subito un messaggio in cui dichiara il numero PID del figlio, quindi si mette in attesa della sua conclusione. Il valore restituito dal processo figlio confluisce nella variabile ***status***, ma deve essere interpretato attraverso la macroistruzione ***WEXITSTATUS()***. Il risultato prodotto a video dal programma di esempio mostrato è molto simile al testo seguente:

```
Ho avviato il processo 6138.
ciao!
Il processo 6138 si è concluso restituendo il valore 7.
```

68.3.4 Adozione dei processi

«

I sistemi Unix e di conseguenza lo standard POSIX, seguono una convenzione nella numerazione dei processi: il kernel è il processo zero ed è implicito; il processo numero uno è '**`init`**' (o altro in situazioni particolari) e ha il compito di essere quello che genera tutti gli altri.

Quando un processo termina di funzionare, i suoi processi figli vengono affidati a '**`init`**' (o comunque a quel processo che si trova ad avere il numero uno).

Listato 68.25. Per provare il codice attraverso un servizio *pastebin*: http://ideone.com/Wq5je .

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
//----------------------------------------------------------------
int
main (void)
{
    pid_t  pid;
    //
    pid = fork ();
    if (pid == 0)
      {
        printf ("Io sono il figlio della biforcazione e "
                "il mio genitore ha il numero "
                "PID %i.\n", (int) getppid ());
        sleep (2);
        printf ("Sono passati due secondi e il mio "
                "genitore ha il numero PID %i.\n",
                (int) getppid ());
        exit (0);
      }
    else
      {
        printf ("Io sono il processo %i e ho avviato una "
                "biforcazione di me stesso, la quale ha "
                "ottenuto il numero PID %i, ma adesso "
                "concludo il mio funzionamento.\n",
                (int) getpid (), pid);
      }
    return (0);
}
```

L'esempio appena mostrato dovrebbe chiarire questo fatto: quando il processo figlio ha superato l'attesa di due secondi, il suo genitore ha già smesso di funzionare, e in quel momento è già stato adottato dal processo numero uno. I messaggi prodotti dal programma sono come quelli seguenti:

```
Io sono il processo 6602 e ho avviato una biforcazione ↩
↪di me stesso, la quale ha ottenuto il numero ↩
↪PID 6603, ma adesso concludo il mio funzionamento.
Io sono il figlio della biforcazione e il mio genitore ↩
↪ha il numero PID 6602.
Sono passati due secondi e il mio genitore ha il ↩
↪numero PID 1.
```

68.3.5 Gli zombie

«

Quando un processo elaborativo conclude il suo funzionamento, per qualunque motivo e in qualunque modo sia, si presume che debba restituire un valore al proprio genitore. Come descritto in precedenza, la funzione ***wait()*** consente a un genitore di recepire la conclusione di un suo processo figlio, ottenendo anche il valore restituito. Tuttavia, non è detto che un genitore sia sempre lì pronto a recepire la conclusione di un proprio figlio, pertanto, i processi conclusi continuano a rimanere annotati nel sistema, fino a quando le loro informazioni devono rimanere disponibili. Un processo concluso, ma in attesa di essere eliminato, è noto come «zombie».

La conclusione di un processo produce automaticamente l'invio di un segnale '**`SIGCHLD`**' al genitore. Questo segnale, in particolare, se non viene intercettato, produce l'eliminazione dei processi figli defunti. Tuttavia potrebbe essere utilizzato da un genitore per intervenire contestualmente e recepire la conclusione di un processo figlio,

senza rimanere in attesa con la funzione ***wait()*** per questo, come nell'esempio seguente:

Listato 68.27. Per provare il codice attraverso un servizio *pastebin*: http://ideone.com/JYqHo .

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
//----------------------------------------------------------
void signal_handler (int signal)
{
    int   status;
    pid_t pid;
    if (signal == SIGCHLD)
      {
        pid = wait (&status);
        printf ("Il processo %i si è concluso restituendo "
                "il valore %x.\n",
                pid, WEXITSTATUS (status));
      }
}
//----------------------------------------------------------
int
main (void)
{
    pid_t  pid;
    //
    signal (SIGCHLD, signal_handler);
    //
    pid = fork ();
    if (pid == 0)
      {
        sleep (1);
```

```
            printf ("ciao!\n");
            exit (7);
          }
        printf ("Ho avviato il processo %i.\n", pid);
        //
        sleep (60);
        //
        return (0);
}
```

Come già chiarito, quando un processo muore, i suoi figli vengono adottati automaticamente da '**`init`**', o comunque dal processo numero uno. In questa circostanza, però, '**`init`**' riceve anche il segnale '**`SIGCHLD`**', perché i processi adottati potrebbero trovarsi già nello stato di «zombie», ovvero in attesa di essere considerati per poter morire definitivamente.

68.3.6 Variabili di ambiente

«

Una caratteristica dei programmi di un sistema Unix, e quindi POSIX, è la disponibilità di quelle che sono note come variabili di ambiente. Va osservato che questo concetto non è presente nel linguaggio C puro e semplice; inoltre, per la stessa ragione, il prototipo della funzione ***main()*** diventa più articolato rispetto a quello di un programma C comune:

```
int main (int argc, char *argv[], char *envp[]);
```

I parametri ***argc*** e ***argv[]*** sono gli stessi, già conosciuti nel linguaggio C, con l'accortezza di avere l'elemento ***argv[argc]*** pari al puntatore nullo ('**`NULL`**'). Il parametro ***envp[]*** è inteso come un array di

stringhe, il cui contenuto deve avere la forma '***nome*=*valore***' e l'ultimo elemento, anche in questo caso, deve essere un puntatore nullo (per poter riconoscere la sua conclusione).

In pratica, l'array ***envp[]*** diventa il veicolo per le variabili di ambiente da fornire al programma che si vuole avviare.

Ma la questione non si esaurisce così, perché per motivi storici l'array di stringhe che descrivono le variabili di ambiente è accessibile anche attraverso una variabile globale (esterna), denominata ***environ***. In tal modo, anche se la funzione ***main()*** non fosse provvista del parametro ***envp[]***, sarebbe comunque possibile accedere alle stringhe delle variabili di ambiente.

```
extern char **environ;
```

Per leggere e modificare ciò che rappresenta le variabili di ambiente, si usano poi delle funzioni apposite, i cui prototipi appaiono nel file di intestazione '`stdlib.h`'. Queste funzioni hanno in comune un nome terminante per «env», come ***setenv()***, ***unsetenv()*** e ***putenv()***.

68.3.7 Variabili di ambiente e avvio di un programma

Quando si avvia un nuovo programma, attraverso una delle funzioni ***exec...()***, questo ottiene un insieme di variabili di ambiente, ereditandole dal processo originario (che viene rimpiazzato), oppure attraverso una dichiarazione esplicita. Ciò dipende da quale funzione ***exec...()*** viene usata effettivamente. La funzione da cui poi hanno origine le altre della famiglia ***exec...()*** è ***execve()***:

```
int execve (const char *path, char *const argv[],
            char *const envp[]);
```

Logicamente, il primo parametro (***path***) rappresenta il percorso del programma da avviare, mentre gli altri due corrispondono agli array di stringhe che di norma hanno lo stesso nome nel prototipo della funzione ***main()***.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
//----------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
    pid_t  pid;
    char  *exec_argv[3];
    char  *exec_envp[3];
    //
    exec_envp[0] = "PATH=/bin:/usr/bin:/sbin:/usr/sbin";
    exec_envp[1] = "CONSOLE=/dev/console";
    exec_envp[2] = NULL;
    //
    exec_argv[0] = "./mio_prog";
    exec_argv[1] = "-x";
    exec_argv[2] = NULL;
    //
    pid = fork ();
    if (pid == 0)
      {
        execve (exec_argv[0], exec_argv, exec_envp);
        perror (NULL);
        exit (1);
      }
```

```
    return (0);
}
```

L'esempio mostra la costruzione degli array contenenti le variabili di ambiente e gli argomenti del programma da avviare.

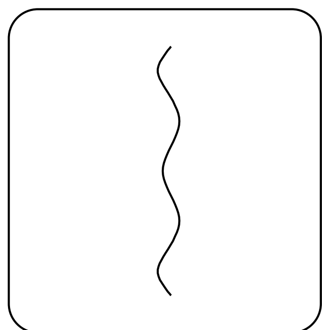
68.4 Nozioni sui thread POSIX

«

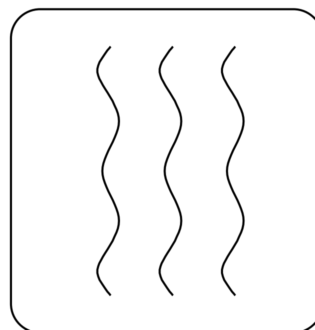
Un programma comune si traduce solitamente in un solo flusso di controllo (o flusso elaborativo), ovvero in un solo ***thread***, nel senso che il procedimento esecutivo è unico, dall'avvio alla conclusione del processo. Un programma più sofisticato, potrebbe gestire gli stessi dati attraverso più flussi di controllo concorrenti e in tal caso si dice che questo utilizza più thread. Pertanto, non va confuso il concetto di processo elaborativo con il flusso di controllo o thread, perché i thread di un processo condividono la stessa memoria, mentre i processi elaborativi, tra di loro, hanno aree di memoria indipendenti.

Il termine inglese thread si traduce letteralmente come «filetto», pertanto viene rappresentato frequentemente in questo modo.

processo elaborativo
a thread singolo



processo elaborativo
con più thread simultanei



La simultaneità di esecuzione dei thread può essere simulata, attraverso la suddivisione del tempo di CPU, oppure può essere anche reale, quando l'elaboratore dispone di più CPU. Tuttavia anche quando si dispone di una sola CPU, l'organizzazione corretta di un programma in più thread può migliorarne le prestazioni.

Lo standard POSIX definisce alcune funzioni per la gestione dei thread, per le quali è necessario includere il file di intestazione '**pthread.h**' (dove la «p» sta per «POSIX»).

In un sistema GNU, o comunque quando si utilizza il compilatore GCC con la libreria dei sistemi GNU, per l'utilizzo delle funzioni che consentono di gestire i thread POSIX, è necessario includere esplicitamente la libreria '**pthread**', con l'opzione '**-lpthread**'. In pratica, per compilare gli esempi di questo capitolo si usano comandi del tipo:

$ **cc -Wall -lpthread -o** ***file_eseguibile file_sorgente_c*** [*Invio*]

68.4.1 Numero identificativo del thread

«

Ogni thread ha un proprio numero identificativo, rappresentato attraverso un tipo di dati apposito, denominato '**pthread_t**'. Quando si crea un thread occorre fare riferimento a una variabile di tipo '**pthread_t**', in modo tale che questa sia aggiornata con il numero corretto; successivamente, per ricondurre un thread al flusso principale del processo elaborativo, si utilizza nuovamente quel numero per poterlo individuare.

L'esempio seguente crea la variabile scalare ***mio_thread***, per annotare il numero di un thread:

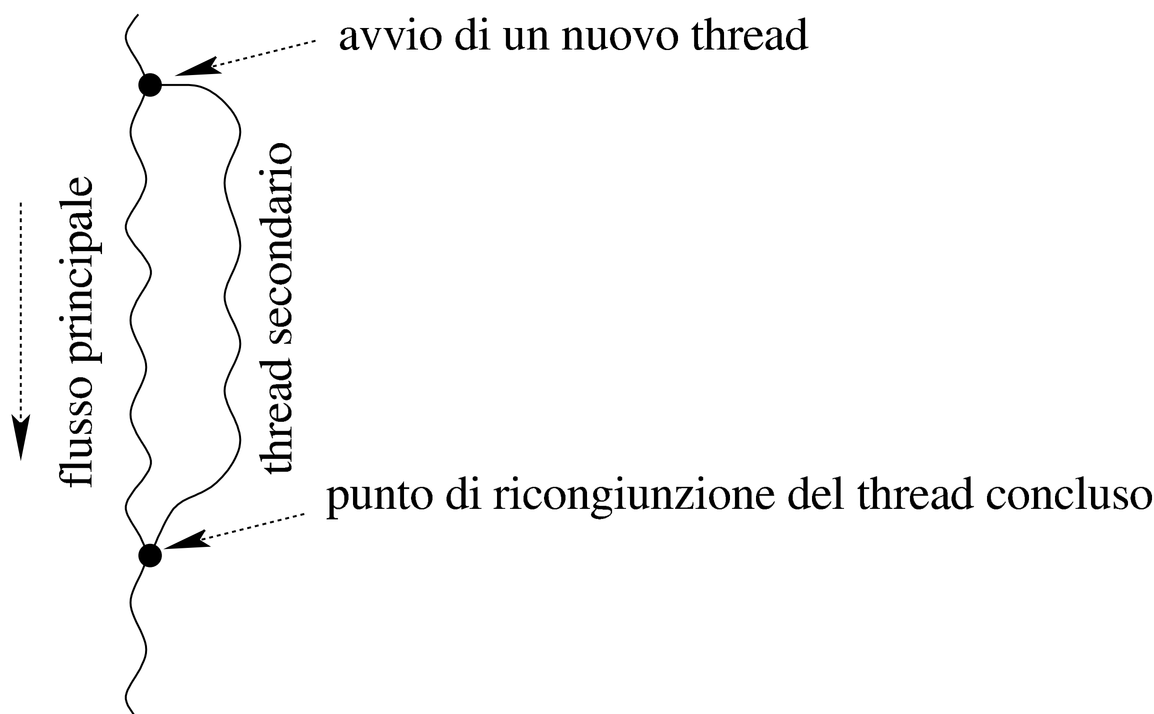
```
pthread_t mio_thread;
```

L'esempio successivo, invece, predispone l'array ***miei_thread[]*** per annotare il numero di identificazione di un massimo di cinque thread:

```
pthread_t miei_thread[5];
```

68.4.2 Creazione e conclusione di un thread aggiuntivo

Un programma ha sempre almeno un thread, ovvero quello principale, la cui creazione è implicita. Tutti gli altri thread che si vogliono gestire vanno creati appositamente: si tratta di fare in modo che una certa funzione sia eseguita senza attenderne la sua conclusione. Ma a un certo punto del flusso principale del programma, è necessario formalizzare la conclusione dei thread aggiuntivi (e se non sono ancora terminati occorre attendere che lo siano effettivamente).



68.4.3 Caratteristiche della funzione che costituisce un thread aggiuntivo

La creazione di un nuovo thread coincide con l'avvio di una funzione senza attendere la sua conclusione. Tale funzione deve però avere una forma precisa: riceve esattamente un argomento, costituito da un puntatore indefinito ('**`void *`**'), e restituisce un valore, costituito da un puntatore indefinito.

```
void *funzione (void *arg);
```

In pratica, per passare degli argomenti a una funzione di questo tipo, si predispone una struttura con tutto ciò che serve e se ne passa il puntatore; d'altro canto, la funzione deve essere in grado di estrapolare i dati dalla struttura. Come si comprende, tale funzione ha anche difficoltà a restituire un valore, perché può solo produrre un puntatore a qualcosa che deve risultare già definito prima della sua chiamata.

Per comprendere la cosa viene proposto un programma estremamente banale, in cui la funzione ***function()*** si limita a mostrare ripetutamente un certo carattere, in base ai dati forniti attraverso il riferimento a una struttura.

Listato 68.33. Per provare il codice attraverso un servizio *pastebin*: http://codepad.org/2ezjFCNy , http://ideone.com/xK3xC .

```
#include <stdio.h>

struct Arguments {
    char x;
    int  max;
```

```
};

void *
function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    for (i = 0; i < arg->max; i++)
      {
        printf ("%c", arg->x);
      }
    return NULL;
}

int
main (void)
{
    struct Arguments arg = {'x', 10};
    function (&arg);
    printf ("\n");
    return (0);
}
```

Come si vede, la funzione deve sapere come si articola la struttura, per poter accedere ai dati che questa contiene. Generalmente, come nel caso dell'esempio, in una funzione di questo tipo non si restituisce alcunché.

68.4.4 Avvio di thread separati e fusione successiva

«

Per comprendere il meccanismo di avvio di un thread separato e della sua fusione successiva, viene proposto un esempio molto semplice, con cui si mostrano solo i passaggi indispensabili. Per la preci-

sione, oltre al flusso principale, vengono avviati tre thread ulteriori, attraverso la stessa funzione. Nell'esempio, la funzione usata per avviare i thread, riceve gli argomenti tramite una struttura articolata nello stesso modo già visto nella sezione precedente. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-thread-1.c* :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct Arguments {
    char x;
    int  max;
};

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10);

    for (i = 0; i < (arg->max * m); i++)
      {
        if ((i % m) == 0)
          {
            printf ("%c", arg->x);
            fflush (stdout);
          }
      }
    return NULL;
}

int
```

```
main (void)
{
    pthread_t pthread_1;
    pthread_t pthread_2;
    pthread_t pthread_3;

    struct Arguments arg_1 = {'a', 12};
    struct Arguments arg_2 = {'b',  6};
    struct Arguments arg_3 = {'c',  3};

    int status_1;
    int status_2;
    int status_3;

    srand (1234567);

    status_3 = pthread_create (&pthread_3, NULL,
                               pthread_function, &arg_3);
    status_2 = pthread_create (&pthread_2, NULL,
                               pthread_function, &arg_2);
    status_1 = pthread_create (&pthread_1, NULL,
                               pthread_function, &arg_1);

    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella creazione "
                         "dei \"thread\"!\n");
        abort ();
      }

    status_1 = pthread_join (pthread_1, NULL);
    status_2 = pthread_join (pthread_2, NULL);
    status_3 = pthread_join (pthread_3, NULL);
```

```
    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella fusione "
                         "dei \"thread\"!\n");
        abort ();
      }

    return (0);
}
```

All'inizio della funzione ***main()*** si può vedere la dichiarazione di tre variabili di tipo '**`pthread_t`**', ognuna delle quali viene usata per annotare un numero di identificazione di un thread. Con la chiamata alla funzione ***pthread_create()*** vengono avviati i thread, indicando il riferimento alla variabile da usare per annotare il thread rispettivo, il riferimento alla funzione da avviare e il riferimento alla struttura contenente gli argomenti per tale funzione. Il prototipo seguente è semplificato, per facilitarne la lettura:

```
int pthread_create (pthread_t *tid, pthread_attr_t *attr,
                    void *(*funzione) (void *),
                    void *argomenti);
```

Generalmente, si usa la funzione ***pthread_create()*** senza specificare attributi particolari per il thread che si vuole creare, pertanto si utilizza semplicemente come secondo argomento il valore '**`NULL`**'.

La funzione ***pthread_create()*** restituisce un valore intero, dove lo zero manifesta il successo dell'operazione, mentre un valore differente indica un problema, decodificabile attraverso il confronto con delle macro-variabili prestabilite. La funzione che avvia il thread for-

nisce il numero dello stesso modificando il contenuto della variabile a cui si riferisce il puntatore fornito come primo argomento.

Una volta accertato che i thread sono stati creati con successo (diversamente il programma termina di funzionare, attraverso la chiamata della funzione ***abort()***), non essendoci altro da fare in questo esempio, viene richiesto di attendere la loro conclusione, attraverso la chiamata della funzione ***pthread_join()***. Tale funzione richiede di indicare il numero del thread di cui si vuole attendere la conclusione, oltre a un puntatore, utile per raggiungere il valore che potrebbe essere restituito dalla funzione che costituiva il thread.

```
int pthread_join (pthread_t tid, void **valore);
```

In pratica, la funzione ***pthread_join()*** sospende l'esecuzione del flusso principale, fino a quando il thread individuato dal numero fornito come primo argomento si conclude (si osservi che in questo caso il numero del thread viene fornito come valore e non più come puntatore). Se il thread non deve produrre alcun risultato utile, il secondo argomento di ***pthread_join()*** può essere il valore nullo ('**`NULL`**'), altrimenti si deve indicare un puntatore generico, a una variabile che contiene a sua volta un puntatore: tale variabile deve essere quella usata dalla funzione che costituiva il thread per porvi al suo interno il puntatore al risultato dell'elaborazione.

La funzione ***pthread_join()*** restituisce un valore intero, dove lo zero indica il successo dell'operazione, mentre un valore diverso rappresenta un problema, individuabile attraverso il confronto con delle macro-variabili prestabilite.

Il programma di esempio, dopo la fusione dei thread e dopo il con-

trollo dell'esito di tale fusione, si conclude semplicemente. Va osservato che la fusione dei thread è necessaria anche in questo caso, perché il programma non può concludersi (attraverso la fine del flusso principale) prima che tutti i thread accessori siano stati fusi.

La funzione utilizzata per i thread dell'esempio, ovvero ***pthread_function()***, trova un numero casuale abbastanza grande e lo usa per eseguire un ciclo per un numero molto elevato di volte. Al primo ciclo, e poi anche ogni volta che l'indice del ciclo risulta divisibile per il numero casuale trovato, mostra una lettera sullo schermo. A questo proposito, va osservato l'uso della funzione ***fflush()*** per garantire che la lettera emessa attraverso lo standard output venga visualizzata subito, senza rimanere in attesa nella memoria tampone.

I thread dell'esempio vengono avviati con insiemi di dati differenti, in modo che: il thread ***pthread_3*** emetta la lettera «c» per tre volte, il thread ***pthread_2*** emetta la lettera «b» per sei volte e che il thread ***pthread_1*** emetta la lettera «a» per dodici volte.

Il risultato visibile sullo schermo assomiglia a una sequenza come questa:

```
cbaaaaaabcaaaabacabbb
```

68.4.5 Conflitto nell'accesso ai dati

«

Quando un thread opera su dati propri (a cui nessun altro thread, nemmeno quello principale, accede in scrittura), tutto fila liscio senza preoccupazioni. Ma la realtà richiede generalmente che i thread si scambino dei dati, pertanto, quando si aggiorna un'informazione, occorre un modo per escludere gli altri thread dall'interferire.

Nell'esempio successivo si crea volutamente una situazione di conflitto tra alcuni thread che modificano simultaneamente una variabile, denominata ***global***, il cui scopo sarebbe quello di contare i caratteri mostrati sullo schermo. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-thread-2.c* .

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct Arguments {
    char x;
    int  max;
};

int global;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10);
    int g;

    for (i = 0; i < (arg->max * m); i++)
      {
        g = global;
        if ((i % m) == 0)
          {
            printf ("%c", arg->x);
            fflush (stdout);
            g++;
            global = g;
```

```
            }
        }
    return NULL;
}

int
main (void)
{
    pthread_t pthread_1;
    pthread_t pthread_2;
    pthread_t pthread_3;

    struct Arguments arg_1 = {'a', 12};
    struct Arguments arg_2 = {'b',  6};
    struct Arguments arg_3 = {'c',  3};

    int status_1;
    int status_2;
    int status_3;

    global = 0;

    srand (1234567);

    status_3 = pthread_create (&pthread_3, NULL,
                               pthread_function, &arg_3);
    status_2 = pthread_create (&pthread_2, NULL,
                               pthread_function, &arg_2);
    status_1 = pthread_create (&pthread_1, NULL,
                               pthread_function, &arg_1);

    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella creazione "
```

```
                            "dei \"thread\"!\n");
            abort ();
          }

        status_1 = pthread_join (pthread_1, NULL);
        status_2 = pthread_join (pthread_2, NULL);
        status_3 = pthread_join (pthread_3, NULL);

        if ((status_1 + status_2 + status_3) != 0)
          {
            fprintf (stderr, "Errore nella fusione "
                            "dei \"thread\"!\n");
            abort ();
          }

        printf ("\n");
        printf ("La variabile globale ha raggiunto "
                "il valore %i.\n", global);

        return (0);
}
```

Provando a eseguire il programma di esempio, si potrebbero osservare messaggi molto simili a quelli seguenti:

```
cbaaaaaacbaaaaabcabbb
La variabile globale ha raggiunto il valore 19.
```

In questo caso la variabile globale che viene modificata dalla funzione ***pthread_function()*** ha raggiunto solo il valore 19, mentre il valore atteso sarebbe di 21 (essendo visualizzati 21 caratteri sullo schermo). Naturalmente può succedere che il valore ottenuto dalla variabile sia corretto, ma non ci si può contare, perché non è possibile

prevedere la sequenza effettiva delle operazioni.

Naturalmente, si può migliorare la funzione ***pthread_function()*** per ridurre al minimo la possibilità di accavallamenti tra le attività dei vari thread, ma anche così non si può avere la garanzia di evitare i conflitti:[2]

```
void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10);
    int g;
    //
    for (i = 0; i < (arg->max * m); i++)
      {
        if ((i % m) == 0)
          {
            printf ("%c", arg->x);
            fflush (stdout);
            g = global;
            g++;
            global = g;
          }
      }
    return NULL;
}
```

68.4.6 Accesso alle risorse in modo mutualmente esclusivo

Per accedere simultaneamente, in modo ordinato, a dati condivisi, occorre definire dei ***mutex***, ovvero delle variabili che hanno il ruolo di «lucchetto» per definire un accesso mutualmente esclusivo a

una certa area di dati. In altri termini, una volta definita una certa attività da svolgere in modo esclusivo, in un certo insieme di dati, gli si associa una variabile speciale con funzione di mutex (lucchetto mutualmente esclusivo) e prima di entrare nella zona critica che richiede un accesso esclusivo a quell'insieme di dati, si cerca di ottenere tale esclusività con una funzione che interroga e modifica la variabile mutex.

```
...
pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 1000);
    int g;

    for (i = 0; i < (arg->max * m); i++)
      {
        pthread_mutex_lock (&mutex_1);
        g = global;
        if ((i % m) == 0)
          {
            printf ("%c", arg->x);
            fflush (stdout);
            g++;
            global = g;
          }
        pthread_mutex_unlock (&mutex_1);
      }
    return NULL;
}
```

```
...
```

L'estratto di esempio appena mostrato mette in evidenza le modifiche da apportare per gestire il meccanismo di accesso mutualmente esclusivo. In questo caso la porzione di codice da eseguire in modo mutualmente esclusivo va dalla lettura della variabile globale alla sua modifica successiva: nell'esempio le operazioni sono tenute distanti per dimostrare il funzionamento, dato che sarebbe meglio ridurre al minimo il tempo in cui un thread blocca un mutex.

La variabile globale '**`mutex_1`**' viene dichiarata di tipo '**`pthread_mutex_t`**' (presumibilmente si tratta di una struttura) e viene inizializzata attraverso una macro-variabile appropriata alle sue caratteristiche. Successivamente, prima di entrare nella zona critica, il thread deve richiedere l'accesso esclusivo attraverso la funzione ***pthread_mutex_lock()***, specificando il riferimento alla variabile che costituisce il mutex del contesto. Quando il thread ottiene l'accesso esclusivo può riprendere la sua esecuzione e, quando non ha più bisogno di impegnare il mutex, lo libera, con la funzione ***pthread_mutex_unlock()***.

Per completezza viene mostrato il programma di esempio completo. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-thread-3.c* .

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct Arguments {
    char x;
```

```
    int  max;
};

int global;

pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 1000);
    int g;
    //
    for (i = 0; i < (arg->max * m); i++)
      {
        pthread_mutex_lock (&mutex_1);
        g = global;
        if ((i % m) == 0)
          {
            printf ("%c", arg->x);
            fflush (stdout);
            g++;
            global = g;
          }
        pthread_mutex_unlock (&mutex_1);
      }
    return NULL;
}

int
main (void)
{
```

```
    pthread_t pthread_1;
    pthread_t pthread_2;
    pthread_t pthread_3;

    struct Arguments arg_1 = {'a', 12};
    struct Arguments arg_2 = {'b',  6};
    struct Arguments arg_3 = {'c',  3};

    int status_1;
    int status_2;
    int status_3;

    global = 0;

    srand (1234567);

    status_3 = pthread_create (&pthread_3, NULL,
                               pthread_function, &arg_3);
    status_2 = pthread_create (&pthread_2, NULL,
                               pthread_function, &arg_2);
    status_1 = pthread_create (&pthread_1, NULL,
                               pthread_function, &arg_1);

    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella creazione "
                         "dei \"thread\"!\n");
        abort ();
      }

    status_1 = pthread_join (pthread_1, NULL);
    status_2 = pthread_join (pthread_2, NULL);
    status_3 = pthread_join (pthread_3, NULL);
```

```
    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella fusione "
                         "dei \"thread\"!\n");
        abort ();
      }

    printf ("\n");
    printf ("La variabile globale ha raggiunto "
            "il valore %i.\n", global);

    return (0);
}
```

Eseguendo il programma si può osservare che non si creano più accavallamenti nella scrittura della variabile globale e il risultato finale è sempre corretto:

```
cbaaaaaabcaaaaaabcbbb
La variabile globale ha raggiunto il valore 21.
```

Come già descritto nella sezione precedente, è sicuramente meglio ridurre al minimo la zona critica, in modo che anche con l'ausilio delle variabili mutex non sia penalizzata la simultaneità di esecuzione dei thread:

```
void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10000);
    int g;

```

```
    for (i = 0; i < (arg->max * m); i++)
      {
        if ((i % m) == 0)
          {
            printf ("%c", arg->x);
            fflush (stdout);
            pthread_mutex_lock (&mutex_1);
            g = global;
            g++;
            global = g;
            pthread_mutex_unlock (&mutex_1);
          }
      }
    return NULL;
}
```

68.4.7 Accesso esclusivo, ma condizionato

Può darsi che l'accesso esclusivo a una zona critica debba avvenire solo al verificarsi di una certa condizione. In altri termini, può darsi che prima di intervenire effettivamente in un certo insieme di dati, un thread debba attendere che questi siano pronti. Per ottenere questo risultato, generalmente, a fianco della variabili mutex, si associano delle variabili che rappresentano il verificarsi di una certa condizione, da gestire anche queste attraverso funzioni apposite dei thread POSIX.

L'estratto seguente mostra le modifiche importanti agli esempi già apparsi, per produrre una situazione in cui i thread devono attendere il verificarsi di una condizione per procedere con il loro intervento nell'area critica:

```
struct Arguments {
    char x;
    int  max;
    int  delay;
};

int global;

pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond_1  = PTHREAD_COND_INITIALIZER;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10000);
    int g;

    for (i = 0; i < (arg->max * m); i++)
      {
        if ((i % m) == 0)
          {
            pthread_mutex_lock (&mutex_1);
            while (global < arg->delay)
              {
                pthread_cond_wait (&cond_1, &mutex_1);
              }
            printf ("%c", arg->x);
            fflush (stdout);
            g = global;
            g++;
            global = g;
            pthread_cond_broadcast (&cond_1);
```

```
            pthread_mutex_unlock (&mutex_1);
          }
      }
    return NULL;
}
```

Questa volta, la struttura che costituisce gli argomenti della funzione ***pthread_function()*** ha un'informazione in più, che rappresenta un ritardo da inserire prima di iniziare a mostrare i caratteri sullo schermo. In pratica, se la variabile globale ha raggiunto o superato quel tale valore, il thread può procedere con il proprio lavoro, altrimenti deve rimanere in attesa.

Per ottenere questo risultato, la variabile globale '**`cond_1`**' viene dichiarata con il tipo '**`pthread_cond_t`**', allo scopo di poter rappresentare le condizioni dei thread, e viene inizializzata con una macrovariabile appropriata alle sue caratteristiche effettive. Il thread, prima cerca di ottenere un accesso esclusivo, quindi, se lo ottiene, inizia un ciclo in attesa del verificarsi della condizione, richiamando ripetutamente la funzione ***pthread_cond_wait()***, con il riferimento alla variabile della condizione e a quella del mutex.

La chiamata della funzione ***pthread_cond_wait()*** fa sì che il thread che aveva ottenuto l'accesso esclusivo venga messo in pausa, a vantaggio di un altro che può così ottenere l'accesso esclusivo alla zona critica. La pausa in cui si trova il primo thread può terminare nel momento in cui viene usata la funzione ***pthread_cond_broadcast()***, con il riferimento alla condizione che aveva prodotto la sospensione e poi anche la funzione ***pthread_mutex_unlock()***.

Il thread che era stato messo in pausa dalla funzione

pthread_cond_wait(), riprende quando tale funzione ha riottenuto l'accesso esclusivo in base alla propria variabile mutex.

Logicamente, occorre fare attenzione a non creare una situazione in per cui tutti i thread si mettono in pausa per qualcosa che non si verifica.

Segue il programma di esempio, completo di tutte le sue parti. Il file dovrebbe essere disponibile presso *allegati/c/esempio-posix-thread-4.c* .

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct Arguments {
    char x;
    int  max;
    int  delay;
};

int global;

pthread_mutex_t mutex_1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond_1  = PTHREAD_COND_INITIALIZER;

void *
pthread_function (void *arguments)
{
    struct Arguments *arg = (struct Arguments *) arguments;
    long int i;
    long int m = (rand () / 10000);
    int g;

    for (i = 0; i < (arg->max * m); i++)
```

```
      {
        if ((i % m) == 0)
          {
            pthread_mutex_lock (&mutex_1);
            while (global < arg->delay)
              {
                pthread_cond_wait (&cond_1, &mutex_1);
              }
            printf ("%c", arg->x);
            fflush (stdout);
            g = global;
            g++;
            global = g;
            pthread_cond_broadcast (&cond_1);
            pthread_mutex_unlock (&mutex_1);
          }
      }
    return NULL;
}

int
main (void)
{
    pthread_t pthread_1;
    pthread_t pthread_2;
    pthread_t pthread_3;

    struct Arguments arg_1 = {'a', 12,  0};
    struct Arguments arg_2 = {'b',  6,  5};
    struct Arguments arg_3 = {'c',  3, 10};

    int status_1;
    int status_2;
    int status_3;
```

```
    global = 0;

    srand (1234567);

    status_3 = pthread_create (&pthread_3, NULL,
                               pthread_function, &arg_3);
    status_2 = pthread_create (&pthread_2, NULL,
                               pthread_function, &arg_2);
    status_1 = pthread_create (&pthread_1, NULL,
                               pthread_function, &arg_1);

    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella creazione "
                         "dei \"thread\"!\n");
        abort ();
      }

    status_1 = pthread_join (pthread_1, NULL);
    status_2 = pthread_join (pthread_2, NULL);
    status_3 = pthread_join (pthread_3, NULL);

    if ((status_1 + status_2 + status_3) != 0)
      {
        fprintf (stderr, "Errore nella fusione "
                         "dei \"thread\"!\n");
        abort ();
      }

    printf ("\n");
    printf ("La variabile globale ha raggiunto "
            "il valore %i.\n", global);
```

```
    return (0);
}
```

Eseguendo il programma si può ottenere un risultato simile a quello seguente:

```
aaaaabbbbbcbccaaaaaaa
La variabile globale ha raggiunto il valore 21.
```

Nel programma di esempio, il thread associato alla variabile '**`pthread_1`**' può visualizzare subito i suoi caratteri sullo schermo, mentre quello associato a '**`pthread_2`**' deve attendere che sia stato visualizzato il quinto, mentre quello associato a '**`pthread_3`**' deve attendere che sia stato visualizzato il decimo. Naturalmente, se tutti i thread avviati dovessero attendere qualche carattere prima di poter iniziare, questi si bloccherebbero a vicenda, irrimediabilmente; inoltre, lo stesso succederebbe se ci fosse anche un solo thread che deve attendere un valore per la variabile '**`global`**' che non può essere raggiunto senza il proprio apporto.

68.4.8 Osservazioni finali

«

Lo standard POSIX prevede una discreta quantità di funzioni per la gestione dei thread; pertanto quanto descritto in questo capitolo è solo il minimo indispensabile per comprenderne il meccanismo. In modo particolare, va tenuto in considerazione che per l'inizializzazione delle variabili mutex e di quelle che rappresentano le condizioni, si possono usare funzioni apposite che non sono state descritte.

Il fatto che ci siano thread distinti rispetto a quello principale, ha delle implicazioni anche sull'invio dei segnali e sulla possibilità di una loro conclusione anticipata. Pertanto sono disponibili le funzioni

pthread_kill() e ***pthread_exit()***, rivolte specificatamente ai thread (escluso sempre il flusso principale):

```
int pthread_kill (pthread_t tid, int segnale);
```

```
void pthread_exit (void *valore_da_restituire);
```

Infine può essere utile la funzione ***pthread_self()***, per ottenere il numero identificativo del thread attuale:

```
pthread_t pthread_self (void);
```

68.5 I file secondo i sistemi POSIX

Il linguaggio C, puro e semplice, prevede una gestione dei file basilare, attraverso il tipo derivato '**`FILE`**', per cui un file aperto è un «flusso», identificato da un puntatore al tipo '**`FILE`**'. Lo standard dei sistemi Unix comporta un'infrastruttura più articolata per la gestione dei file, al di sotto di quanto già descrive il C, introducendo il concetto di ***descrittore di file***, corrispondente a un numero intero normale positivo. Le funzioni e le macro-variabili principali per l'apertura e il controllo dei file, secondo la mediazione del concetto di descrittore, sono indicati nel file di intestazione '`fcnt.h`' (*file control*), ma per amministrare le caratteristiche dei file, servono le definizioni e le funzioni del file di intestazione '`sys/stat.h`'; inoltre, altre funzioni importanti al riguardo si trovano nel file '`unistd.h`'.

L'apertura di un file, dal punto di vista dei sistemi Unix (e quindi POSIX), implica non solo l'associazione al numero del descrittore, ma anche l'attribuzione di opzioni di funzionamento ed eventualmente un sistema di blocco di porzioni del file. La creazione di un file implica l'attribuzione di permessi, nel rispetto però della maschera dei permessi esistente.

Va osservato che anche i flussi di file standard del linguaggio C, trovano una corrispondenza nello standard POSIX in altrettanti descrittori già assegnati, costituiti precisamente dai primi tre:

Denominazione	flusso di file C	numero del descrittore POSIX
standard input	`stdin`	0
standard output	`stdout`	1
standard error	`stderr`	2

Lo standard POSIX prescrive che i numeri dei descrittori siano assegnati usando sempre il valore libero più piccolo; pertanto, il primo descrittore a essere utilizzato, dato che i primi tre sono impegnati per i flussi standard, è il numero tre e di seguito vanno i successivi.

68.5.1 Apertura e chiusura di un file

«

L'apertura ed eventuale creazione di un file, secondo le convenzioni POSIX, va eseguita utilizzando la funzione ***open()***. Per motivi storici esiste anche la funzione ***creat()*** che però ha meno possibilità di ***open()***, pertanto il suo utilizzo non è indispensabile.

```
int open (const char *file, int oflag[, mode_t mode]);
```

```
int creat (const char *file, mode_t mode);
```

La funzione ***open()*** apre un file, indicato attraverso una stringa che descrive il suo percorso (relativo o assoluto che sia) secondo le convenzioni POSIX e restituisce il numero del suo descrittore; se però restituisce il valore −1, significa che l'operazione non ha avuto successo e di conseguenza è stato modificato il contenuto della variabile ***errno*** (la quale può essere esaminata per determinarne la causa).

Il valore costituito dal parametro ***oflag*** viene ottenuto combinando assieme, con l'operatore OR binario, una serie di macro-variabili definite nel file '`fcntl.h`', tenendo conto che non tutte le combinazioni sono ammissibili simultaneamente. Se si utilizza l'opzione rappresentata dalla macro-variabile ***O_CREAT***, per richiedere la creazione del file, va usato anche il terzo parametro della funzione, con cui si specifica la modalità di creazione dello stesso.

Tabella 68.47. Macro-variabili da utilizzare per combinare il valore del parametro ***oflag***. Di queste macro-variabili, in particolare, ne va scelta una sola e si è obbligati a usarla per specificare la modalità di accesso al file: in sola lettura, in sola scrittura o in entrambi i modi.

Macro-variabile	Significato
`O_RDONLY`	Si richiede l'apertura del file in sola lettura.

Macro-variabile	Significato
`O_WRONLY`	Si richiede l'apertura del file in sola scrittura.
`O_RDWR`	Si richiede l'apertura del file in lettura e scrittura.

Tabella 68.48. Alcune delle macro-variabili da utilizzare per combinare il valore del parametro ***oflag***. Non tutte le combinazioni di queste opzioni sono ammissibili.

Macro-variabile	Significato
`O_APPEND`	Questa opzione fa in modo che la scrittura avvenga sempre in estensione del contenuto già esistente.
`O_CREAT`	Questa opzione richiede la creazione del file, ammesso che non esista già. Nel caso il file sia effettivamente da creare, vale la modalità di creazione specificata con il parametro ***mode***.
`O_TRUNC`	Questa opzione si può associare solo a una richiesta di accesso in scrittura (in sola scrittura o in lettura e scrittura simultaneamente) relativa a un file normale, con lo scopo di azzerarne il contenuto se questo file esiste già.
`O_EXCL`	Questa opzione può essere usata solo in abbinamento a '**`O_CREAT`**' e richiede la creazione del file o il fallimento dell'operazione se questo esiste già, anche nel caso si tratti solo di un collegamento simbolico che punta a un file inesistente.
`O_NOCTTY`	Questa opzione fa sì che se il file indicato corrisponde a un dispositivo di terminale, questo non possa diventare in alcun caso il terminale di controllo abbinato al processo.

Macro-variabile	Significato
`O_NONBLOCK`	Questa opzione richiede di non attendere per la conclusione delle operazioni, ammesso che ciò sia possibile in base al contesto.
`O_SYNC`	Questa opzione richiede che le operazioni di accesso al file avvengano in modo sincrono rispetto all'hardware. Pertanto, in questo modo, le operazioni di scrittura comportano l'attesa che queste siano realizzate effettivamente.

Il parametro ***mode*** riguarda esclusivamente la creazione del file (specificando l'indicatore '**`O_CREAT`**'). In tal caso, si tratta del numero che esprime i permessi da dare al file. Si tratta degli stessi permessi che si indicano con programmi come '**`chmod`**', quando si usa la forma numerica, e si scrivono preferibilmente in base otto. Naturalmente, i permessi indicati vengono poi filtrati attraverso la maschera dei permessi, come se fosse eseguita questa operazione: '**(*mode* & ~*umask*)**'. Se lo si preferisce, al posto di indicare i permessi richiesti, direttamente in forma numerica, ci si può avvalere di macro-variabili dichiarate nel file di intestazione '`sys/stat.h`', come descritto nella tabella successiva.

Tabella 68.49. Macro-variabili per esprimere i permessi da attribuire a un file che si vuole creare.

Macro-variabile	Valore numerico equivalente	Significato
`S_IRWXU`	0700_8	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per l'utente proprietario.

Macro-variabile	Valore numerico equivalente	Significato
S_IRUSR	0400_8	Rappresenta il permesso di lettura per l'utente proprietario.
S_IWUSR	0200_8	Rappresenta il permesso di scrittura per l'utente proprietario.
S_IXUSR	0100_8	Rappresenta il permesso di esecuzione o attraversamento per l'utente proprietario.
S_IRWXG	0070_8	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per il gruppo proprietario.
S_IRGRP	0040_8	Rappresenta il permesso di lettura per il gruppo proprietario.
S_IWGRP	0020_8	Rappresenta il permesso di scrittura per il gruppo proprietario.
S_IXGRP	0010_8	Rappresenta il permesso di esecuzione o attraversamento per il gruppo proprietario.
S_IRWXO	0007_8	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per gli altri utenti.
S_IROTH	0004_8	Rappresenta il permesso di lettura per gli altri utenti.
S_IWOTH	0002_8	Rappresenta il permesso di scrittura per gli altri utenti.
S_IXOTH	0001_8	Rappresenta il permesso di esecuzione o attraversamento per gli altri utenti.
S_ISUID	4000_8	Rappresenta l'attivazione del bit S-UID.
S_ISGID	2000_8	Rappresenta l'attivazione del bit S-GID.

Macro-variabile	Valore numerico equivalente	Significato
`S_ISVTX`	1000_8	Rappresenta l'attivazione del bit Sticky.

Come già accennato, la funzione ***creat()*** non è più indispensabile e può essere sostituita da ***open()***, usata nel modo seguente:

```
open (file, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

Segue un esempio molto semplice in cui si apre un file in scrittura, specificando che se non esiste già, questo va creato, con tutti i permessi che la maschera dei permessi esistente consenta di attribuire. In caso di errore, il contenuto della variabile ***errno*** viene considerato con l'aiuto della funzione ***perror()***. Il file dell'esempio dovrebbe essere disponibile presso *allegati/c/esempio-posix-fcntl-open.c* .

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int
main (void)
{
    const char *file = "/tmp/test";
    int fdn;

    fdn = open (file, O_WRONLY|O_CREAT, 07777);

    if (fdn >= 0)
      {
        printf ("Aperto il file \"%s\", ", file);
```

```
          printf ("associandolo al descrittore %i.\n", fdn);
          close (fdn);
        }
      else
        {
          printf ("Non è possibile aprire il file \"%s\"!\n",
                  file);
          perror (NULL);
        }

      return (0);
  }
```

La funzione ***open()*** richiede l'inclusione del file '`fcntl.h`', nel quale sono dichiarate anche la macro-variabili ***O_WRONLY*** e ***O_CREAT***; ma per la funzione ***close()*** è necessario includere anche il file '`unistd.h`'.

Come si vede, i permessi da attribuire al file che venisse creato sono tutti quelli disponibili (7777_8). Eventualmente, aggiungendo anche l'inclusione del file '`sys/stat.h`', sarebbe possibile indicare tale richiesta attraverso macro-variabili convenzionali:

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

    ...
    fdn = open (file, O_WRONLY|O_CREAT,
                S_IRUSR|S_IWUSR|S_IRGRP);
    ...
```

In questo esempio, però, i permessi richiesti sono minori, corrispondenti al numero 0640_8.

Va osservato che la funzione ***open()*** può aprire ogni tipo di file, ma può creare solo dei file «normali». Per creare directory e altri tipi di file speciali si usano funzioni apposite. D'altro canto, per attribuire a un file dei permessi, è possibile usare la funzione ***chmod()*** e non è strettamente necessario occuparsene nel momento della creazione.

La funzione ***close()***, già mostrata nell'esempio, ha un prototipo molto semplice: richiede l'indicazione del descrittore del file da chiudere e restituisce zero se tutto va bene, altrimenti produce il valore −1 e aggiorna la variabile ***errno***:

```
int close (int descrittore);
```

68.5.2 Lettura e scrittura

«

Come per le funzioni dello standard C, anche per quelle a livello POSIX si accede al contenuto del file attraverso un indicatore della posizione espresso in byte (con la differenza che non si pone il problema di distinguere tra file di testo e file binari). A ogni descrittore di file sono associate delle informazioni, amministrate in modo trasparente dal sistema operativo, e a queste si accede solo attraverso delle funzioni. Tra queste informazioni si trova anche l'indicatore che consente di determinare la posizione iniziale per la lettura o la scrittura.

A seconda di come viene aperto il file, l'indicatore della posizione che lo riguarda viene inizializzato nel modo più logico, come descritto a proposito della funzione ***open()***. Questo indicatore vie-

ne spostato automaticamente a seconda delle operazioni di lettura e scrittura che si compiono, tuttavia, quando si passa da una modalità di accesso all'altra, è necessario spostare l'indicatore attraverso le istruzioni opportune, in modo da non creare ambiguità.

Per la lettura di un file aperto e qualificato con un descrittore, si può usare la funzione ***read()*** che legge una quantità di byte trattandoli come un array. Si osservi l'esempio seguente:

```
…
char buf[100];
int fdn;
ssize_t dim;
…
fdn = open (…);
…
dim = read (fdn, buf, 100);
…
```

In questo modo si intende leggere 100 byte, collocandoli nell'array ***buf***, con la stessa capacità massima. Naturalmente, non è detto che la lettura abbia successo, o quantomeno non è detto che si riesca a leggere la quantità di elementi richiesta. Il valore restituito dalla funzione rappresenta la quantità di byte letti effettivamente. Se si verifica un qualsiasi tipo di errore che impedisce la lettura, la funzione si limita a restituire -1, mentre lo zero è un risultato valido e indica che la lettura è giunta alla fine del file.

Quando il file viene aperto in lettura, in condizioni normali l'indicatore interno viene posizionato all'inizio del file; quindi, ogni operazione di lettura sposta in avanti il puntatore, in modo che la prossima lettura avvenga a partire dalla posizione immediatamente successiva:

```
...
char buf[100];
int fdn;
ssize_t dim;
...
fdn = open (...);
...
while (1)                // Ciclo senza fine.
  {
    dim = read (fdn, buf, 100);
    if (dim == 0)
      {
        break;           // Termina il ciclo.
      }
    ...
  }
...
```

In questo modo, come mostra l'esempio, viene letto tutto il file a colpi di 100 byte alla volta, tranne l'ultima in cui si ottiene solo quello che resta da leggere.

Analogamente, la scrittura può essere eseguita con la funzione ***write()*** che scrive una quantità di byte trattandoli come un array, nello stesso modo già visto con la funzione ***read()***. Anche in questo caso, la scrittura procede a partire dalla posizione corrente riferita al file.

```
…
    char buf[100];
    int fdn;
    ssize_t dim;
    …
    fdn = open (…);
    …
    dim = write (fdn, buf, 100);
    …
```

L'esempio, come nel caso di ***read()***, mostra la scrittura di 100 byte, prelevati da un array. Il valore restituito dalla funzione è la quantità di elementi che sono stati scritti con successo. Se si verifica un errore la funzione restituisce il valore −1, mentre lo zero è un valore valido.

Anche in scrittura è importante l'indicatore della posizione interna del file. Di solito, quando si crea un file o lo si estende, l'indicatore si trova sempre alla fine. L'esempio seguente mostra lo scheletro di un programma che crea un file, copiando il contenuto di un altro (non viene utilizzato alcun tipo di controllo degli errori).

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
…
int
main (void)
{
    char           buf[1024];
    int            fdn_in;
    int            fdn_out;
    ssize_t        dim;
    …
    fdn_in = open (file, O_RDONLY);
```

```
    …
    fdn_out = open (file, O_WRONLY);
    …
    while (1)                       // Ciclo senza fine.
      {
        dim = read (fdn, buf, 1024);
        if (dim == 0)
          {
            break;                  // Termina il ciclo.
          }
        …
        write (fdn_out, buf, dim);
        …
      }
    …
    close (fdn_in);
    close (fdn_out);
    return 0;
}
```

Seguono i modelli sintattici di ***read()*** e ***write()***, espressi in forma di prototipi di funzione:

```
ssize_t read (int fdn, void *buf, size_t n);
```

```
ssize_t write (int fdn, const void *buf, size_t n);
```

Il tipo di dati '**ssize_t**' rappresenta l'equivalente di '**size_t**', ma con segno, allo scopo di poter rappresentare il valore -1, che indica un esito errato; il tipo '**void**' per l'array in cui vanno scritti o da cui vanno letti i dati, permette l'utilizzo di qualunque tipo per i suoi

elementi, anche se le operazioni di lettura e scrittura operano solo al livello di byte.

68.5.3 Spostamento dell'indicatore interno al file

Lo spostamento diretto dell'indicatore interno della posizione di un file aperto è un'operazione necessaria quando il file è stato aperto simultaneamente in lettura e in scrittura, e da un tipo di operazione si vuole passare all'altro. Per questo si utilizza la funzione ***lseek()***, con la quale è possibile leggere e modificare tale posizione attuale. La posizione e gli spostamenti sono espressi in byte; la variabile usata per rappresentare questi spostamenti è di tipo '`off_t`' (*offset*).

La funzione ***lseek()*** esegue lo spostamento a partire dall'inizio del file, oppure dalla posizione attuale, oppure dalla posizione finale. Per questo utilizza un parametro che può avere tre valori identificati rispettivamente da tre macro-variabili, definite all'interno del file '`stdio.h`': ***SEEK_SET***, ***SEEK_CUR*** e ***SEEK_END***. l'esempio seguente mostra lo spostamento del puntatore, riferito al descrittore di file ***fdn***, in avanti di 10 byte, a partire dalla posizione attuale.

```
…
i = lseek (fdn, 10, SEEK_CUR);
…
```

La funzione ***lseek()*** restituisce la posizione raggiunta all'interno del file, partendo dall'inizio dello stesso, se lo spostamento avviene con successo, altrimenti produce il valore -1.

L'esempio seguente mostra lo scheletro di un programma, senza controlli sugli errori, che, dopo aver aperto un file in lettura e scrittura, lo legge a blocchi di dimensioni uguali, modifica questi blocchi

e li riscrive nel file. Il file dell'esempio dovrebbe essere disponibile presso *allegati/c/esempio-posix-read-write.c* .

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

static const size_t max = 100;      // Dimensione massima del
                                    // record logico.
int
main (void)
{
  const char *file = "/tmp/test"; // File da elaborare.
  char         buf[max];        // Allocazione del record.
  int          fdn;             // Descrittore del file.
  off_t        pos_1;           // Posizione interna al file.
  off_t        pos_2;           // Posizione interna al file.
  ssize_t      dim;             // Dimensione del record letto.

  fdn = open (file, O_RDWR);

  while (1)                     // Ciclo senza fine.
    {
      //
      // Salva la posizione del puntatore interno al file
      // prima di eseguire la lettura.
      //
      pos_1 = lseek (fdn, 0, SEEK_CUR);

      dim = read (fdn, buf, max);

      if (dim == 0)
        {
          break;                // Termina il ciclo.
        }
```

```
        //
        // Salva la posizione del puntatore interno al file
        // dopo la lettura.
        //
        pos_2 = lseek (fdn, 0, SEEK_CUR);
        //
        // Sposta il puntatore alla posizione precedente alla
        // lettura.
        //
        lseek (fdn, pos_1, SEEK_SET);
        //
        // Esegue qualche modifica nei dati, per esempio
        // mette un punto esclamativo all'inizio.
        //
        buf[0] = '!';
        //
        // Riscrive il record modificato.
        //
        write (fdn, buf, dim);
        //
        // Riporta il puntatore interno al file alla posizione
        // corretta per eseguire la lettura successiva
        //
        lseek (fdn, pos_2, SEEK_SET);
    }

    close (fdn);
    return 0;
}
```

Segue il modello sintattico per l'uso della funzione ***lseek()***, espresso attraverso il suo prototipo:

```
off_t lseek (int fdn, off_t spostamento,
             int punto_di_partenza);
```

Il valore dello spostamento, costituito dal secondo parametro, rappresenta una quantità di byte che può essere anche negativa, indicando in tal caso un arretramento dal punto di partenza specificato dal terzo parametro. Il valore restituito da ***lseek()*** è la nuova posizione all'interno del file, espressa a partire dall'inizio dello stesso (pertanto deve trattarsi di un valore maggiore o uguale a zero); se invece si presenta un errore, si ottiene −1 (precisamente si definisce come '**`(off_t) -1`**').

68.5.4 Controllo degli errori

«

Le funzioni descritte, quando si verifica un errore, annotano il numero dell'errore nella variabile globale ***errno*** (il nome ***errno*** dovrebbe essere precisamente un'espressione che si traduce nell'accesso a un'area di memoria condiviso dal programma, distinto in base al thread). Il significato del valore attribuito alla variabile ***errno*** è descritto da macro-variabili definite nel file '`errno.h`', il quale fa già parte dello standard C, ma viene esteso da POSIX.

La lettura della variabile ***errno*** porta alla conoscenza dell'ultimo errore che si è presentato e non è previsto il suo azzeramento automatico; pertanto, occorre accertarsi del verificarsi di un problema, prima di interrogare la variabile, oppure la si deve azzerare prima di chiamare una funzione di cui si vuole verificare l'esito.

Per interpretare l'errore annotato nella variabile ***errno*** e visualizzare direttamente un messaggio attraverso lo standard error, si può usare

la funzione ***perror()***, già descritta nei capitoli sul C:

```
void perror (const char *s);
```

68.6 Il file system Unix e la sua gestione tipica

«

Per comprendere il senso dell'organizzazione della libreria C e di quella POSIX, per quanto riguarda la gestione dei file, è necessario conoscere l'impostazione originale della gestione di un file system in un sistema Unix. A tale riguardo ci sono due livelli: quello del file system, così come viene strutturato nell'unità di memorizzazione e la gestione dei file aperti, a livelli diversi, partendo dall'inode, fino al flusso di file del C, passando per il concetto di descrittore del file.

68.6.1 Il blocco

«

In un file system Unix tradizionale, lo spazio di un'unità di memorizzazione è suddiviso in blocchi di byte, di dimensione pari a un multiplo del settore fisico, ma si tratta comunque di un valore che si ottiene come potenza di 2. Considerato che le unità di memorizzazione comuni hanno settori fisici da 512 byte, il blocco di un tale file system può essere da 1024, 2048, 4096 byte,... La dimensione effettiva di tale blocco dipende però dalle caratteristiche specifiche di quel tipo di file system, tenendo conto che spesso è possibile scegliere la sua dimensione in fase di inizializzazione.

Nel caso del file system Minix, si distingue tra blocchi e zone. La zona è un concetto specifico dei sistemi Minix e, in realtà, la zona di Minix è l'entità del file system che più si avvicina al blocco dei sistemi Unix tradizionali.

68.6.2 Il super blocco

«

Le unità di memorizzazione possono essere organizzate in due modi: con partizioni o senza. Una partizione è a sua volta come un'unità singola, non divisa in partizioni.

L'inizio di un'unità di memorizzazione viene riservato generalmente per il codice di avvio del sistema operativo, tenendo conto che questo vale sia per le unità suddivise in partizioni, sia per quelle non suddivise, sia per le partizioni stesse. Pertanto, nessun file system sovrascrive il primo settore di un'unità, anche se può considerarlo parte della propria gestione.

Dopo lo spazio che viene lasciato per il codice di avvio del sistema operativo (di uno o più settori), si colloca generalmente quello che è noto come ***super blocco***, il quale si può considerare come una tabella riassuntiva delle caratteristiche generali del file system e della sua situazione.

Le informazioni contenute nel super blocco devono consentire di sapere: qual è la dimensione del blocco (ammesso che questa non sia fissa); qual è la dimensione dell'unità in blocchi; dove sono le tabelle che rappresentano gli inode; quanti sono gli inode, quali sono quelli liberi e quali invece sono impegnati; quali sono i blocchi che possono essere utilizzati per i dati (file, directory e altre tabelle per i riferimenti indiretti) e quali invece sono impegnati.

Per conoscere quali sono gli inode e le zone libere o impegnate, si possono usare sistemi diversi. In generale è probabile che si usino mappe di bit (come nel caso di Minix), oppure delle liste. Ma in generale, ciò che consente di sapere e di annotare gli inode e le zone impegnate o libere, fa parte concettualmente del super blocco, anche

se materialmente può trattarsi di strutture di dati separate.

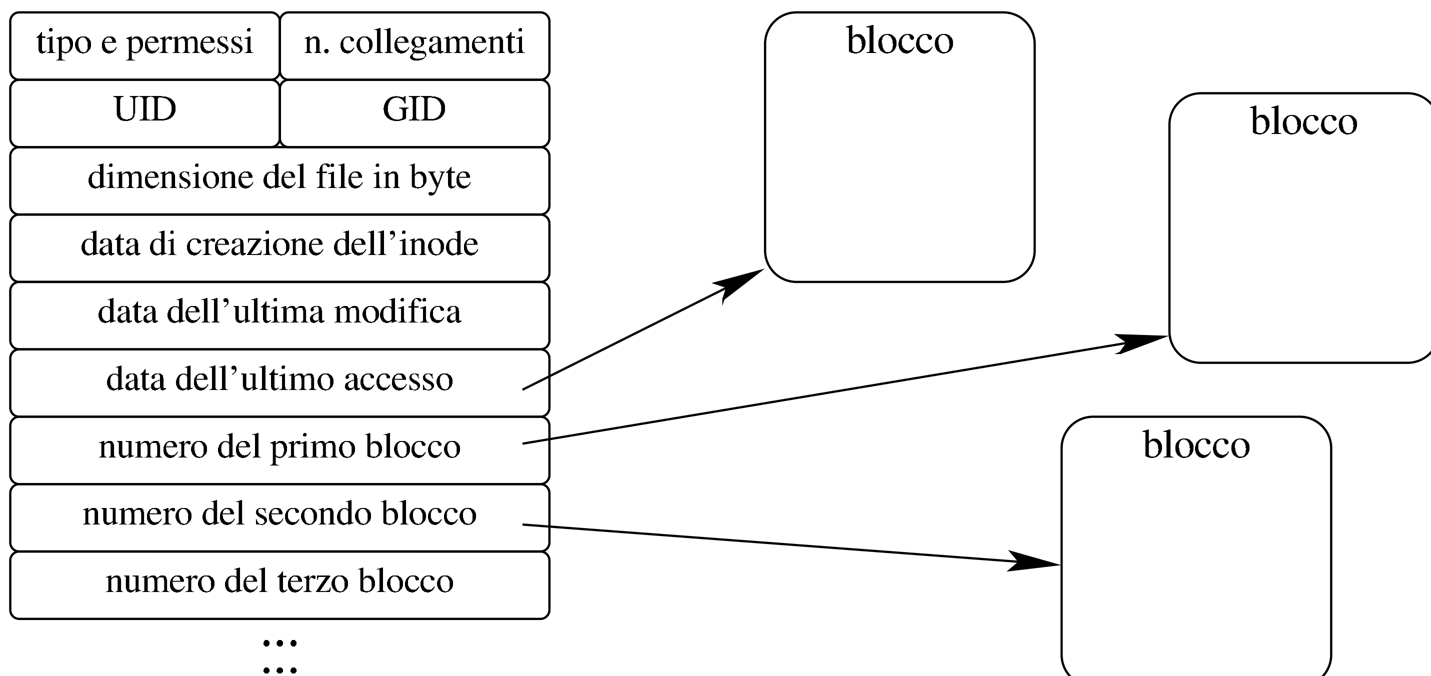
68.6.3 Gli inode

«

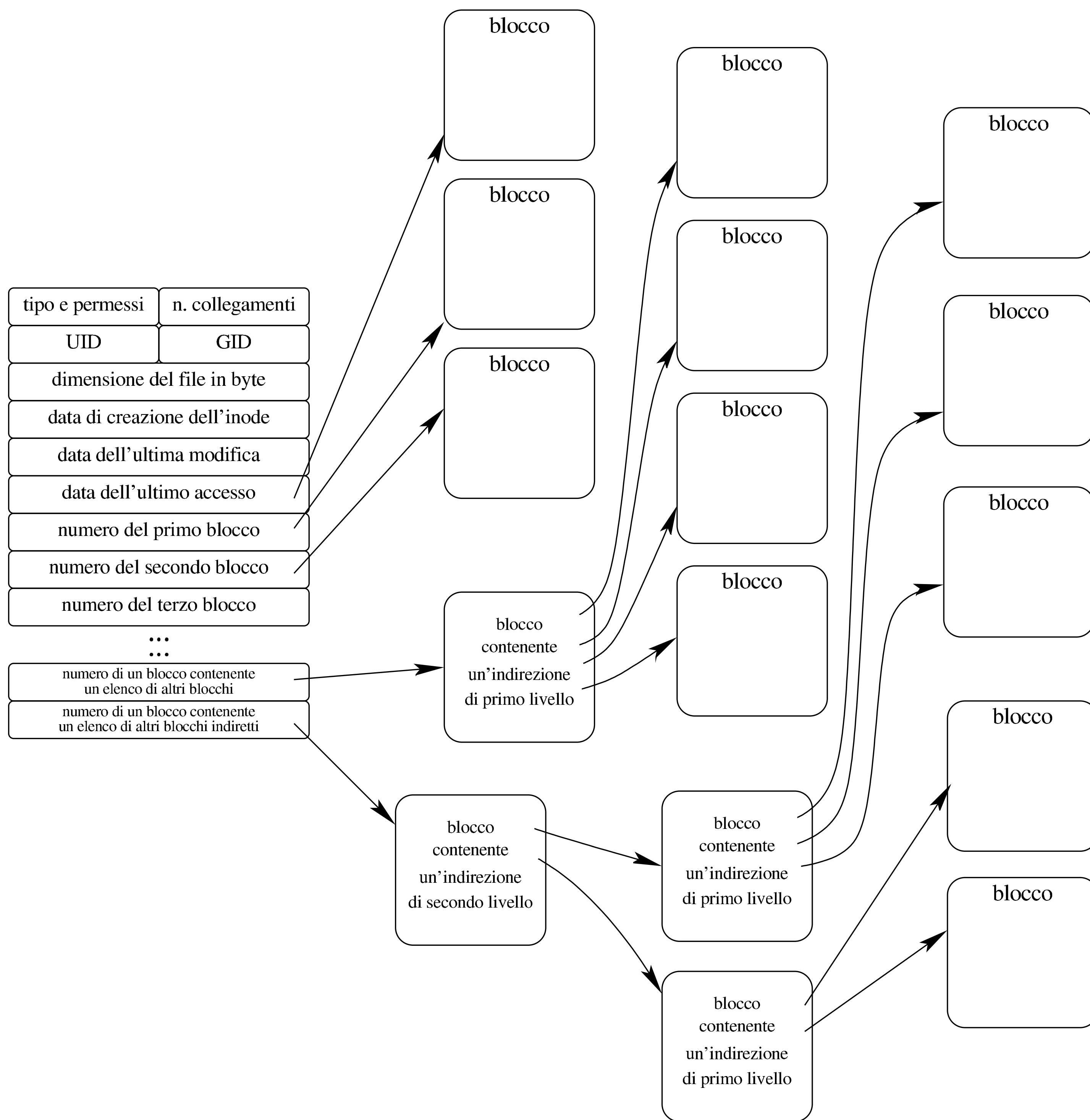
Dopo il super blocco, includendo in questo anche ciò che consente di sapere quali inode e quali blocchi sono impegnati o liberi, si collocano generalmente delle tabelle che rappresentano, ognuna, un inode. Di solito, un inode occupa un sottomultiplo dello spazio di un blocco, in modo da facilitare i calcoli per individuarne la collocazione.

Un inode contiene almeno queste informazioni: il tipo di file e i permessi di accesso, in un valore a 16 bit; il numero UID del proprietario del file; il numero GID del gruppo proprietario; la dimensione del file (o della directory) espressa in byte; le date di accesso, modifica del file e creazione dell'inode; la quantità di riferimenti provenienti dalle directory; una serie di numeri di blocchi occupati dal file o dalla directory.

Il contenuto del file rappresentato dall'inode si articola in blocchi, i quali non sono necessariamente contigui e nemmeno ordinati: è l'insieme dei riferimenti contenuti nell'inode che determina la posizione e l'ordine in cui questi vanno considerati. Inoltre, dato che non è possibile allocare nel file system uno spazio più piccolo di un blocco, è indispensabile l'informazione sulla dimensione del file per sapere quando questo termina nel suo ultimo blocco utilizzato.



Dal momento che deve essere possibile rappresentare file di grandi dimensioni, i primi riferimenti ai blocchi utilizzati sono diretti, come si vede nel disegno appena mostrato, mentre si prevedono generalmente dei riferimenti indiretti, a blocchi che contengono a loro volta i riferimenti di altri blocchi. In tal caso si parla di «indirezione» di primo, secondo ed eventualmente anche di terzo livello.



In questa struttura di collegamenti ai blocchi, va osservato che un file potrebbe non avere allocato tutti blocchi che risulterebbero dalla dimensione riportata. Per esempio, utilizzando blocchi da 1024 byte, un file che risulta essere grande 10240 byte, non è detto che occupi effettivamente 10 blocchi come sembrerebbe. Infatti, la scrittura nel

file potrebbe essere avvenuta specificando una piccola parte verso la fine e in altre posizioni. In pratica, almeno in linea teorica, questo tipo di organizzazione a inode, consente di scrivere il file dove si vuole, sapendo che lo spazio intermedio, se non viene allocato, risulta contenere dei dati con bit a zero.

Nella tabella che rappresenta l'inode, i blocchi non allocati risultano indicati con il numero zero, pertanto, il vero blocco zero non può essere accessibile attraverso gli inode (ma d'altra parte è normale che il blocco zero sia impegnato dal codice di avvio, dal super blocco ed eventualmente dalle tabelle degli inode stessi).

È importante osservare che in un file system Unix non esiste mai l'inode con il numero zero, perché questo valore viene utilizzato per fare riferimento a un errore o comunque a situazioni speciali. Di solito, l'inode numero uno corrisponde alla directory principale, dell'unità presa in considerazione (ovvero di una sua partizione).

68.6.4 La directory

La directory è un file come gli altri, riconoscibile perché, nell'inode, il campo che definisce il tipo e i permessi, riporta l'indicazione relativa. Nei file system tradizionali, il file che rappresenta la directory è formato normalmente da *record* di lunghezza uniforme, in cui si distingue un campo contenente il numero di un inode e un altro contenente il nome di un file.

n. inode	nome
	.
	..
	cat
	cp
	dd

In pratica, si associa il tale inode a un certo nome. Va ricordato che l'inode zero non esiste e che, di norma, l'inode uno è quello della directory radice dell'unità di memorizzazione (o della partizione relativa).

Nel file di intestazione '`limits.h`', la macro-variabile ***NAME_MAX*** rappresenta la quantità minima di caratteri che possono essere usati per i nomi dei file (nelle directory). Questa quantità non include il carattere nullo di terminazione delle stringhe; pertanto, se corrispondesse al valore 14, vorrebbe dire che i nomi possono avere effettivamente 14 caratteri. Ciò avviene a differenza della macro-variabile ***PATH_MAX***, per i percorsi, la quale deve invece includere anche il carattere nullo di terminazione.

La directory si costruisce come descritto, ma rimane il fatto che le prime due voci debbano essere «.» e «..»: la prima corrispondente al riferimento dell'inode della directory stessa; la seconda corrispondente al riferimento dell'inode della directory genitrice (ovvero quella precedente in senso gerarchico), con la variante che la directory radice può solo puntare a se stessa, in ogni caso.

68.6.5 Tabelle del sistema operativo

«

Il file system dei sistemi Unix, oltre che avere una certa forma nella fisicità dell'unità di memorizzazione, ha anche una rappresentazione astratta tradizionale nel sistema operativo, nel modo in cui si prendono in considerazione i file aperti e ciò da cui questi dipendono.

Nella semplificazione dei sistemi tradizionali, si utilizzano delle tabelle per: i super blocchi delle unità innestate; gli inode in corso di utilizzazione; i file aperti; i descrittori dei file aperti; i flussi di file abbinati ai descrittori.

68.6.5.1 Tabella dei super blocchi

«

Per accedere a un file, è necessario poter raggiungere il file system di un certo dispositivo, il quale deve essere stato innestato, ovvero reso disponibile nel file system generale del sistema operativo. Per raccogliere la situazione delle unità innestate serve una tabella, la quale può essere vista come quella dei dispositivi o dei super blocchi.

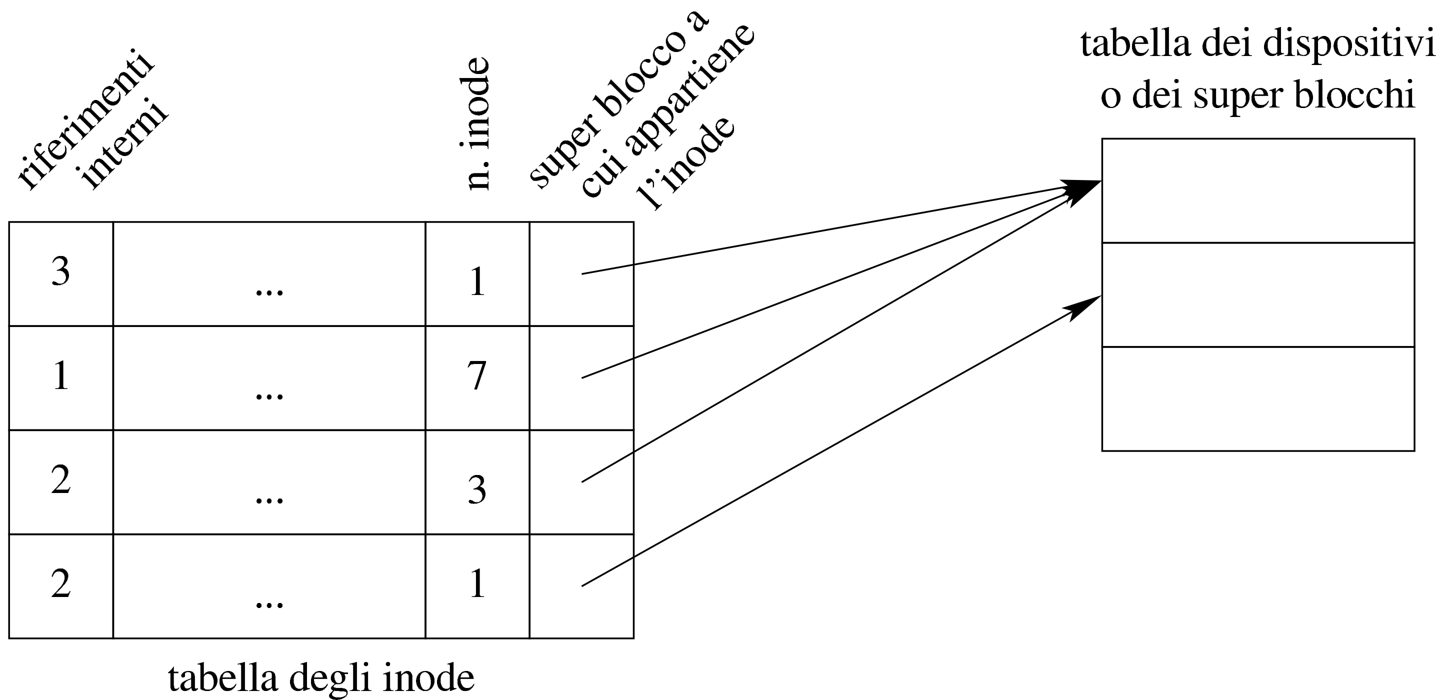
Le voci della tabella dei super blocchi riproducono i super blocchi delle unità innestate, incluse le mappe o le tabelle di utilizzo degli inode e dei blocchi, oltre ad altre informazioni accessorie. Quando di crea o si elimina un file, quando lo si estende e lo si riduce, la voce relativa di tale tabella dei super blocchi va aggiornata anche nell'unità di memorizzazione, per quanto riguarda la situazione di utilizzo degli inode e dei blocchi di dati.

Va osservato che il sistema operativo deve innestare almeno una unità, contenente il file system principale, pertanto, almeno un super blocco deve essere sempre presente nella tabella.

68.6.5.2 Tabella degli inode

Quando si accede a un file per la prima volta, le informazioni relative al suo inode vengono caricate in una voce della tabella degli inode. Il contenuto minimo di questa voce è costituito di norma da tutti i dati dell'inode contenuti nel file system, incluso il numero di questo, aggiungendo il riferimento alla voce che rappresenta il super blocco da cui proviene e la quantità di riferimenti interni (i riferimenti interni non vanno confusi con i collegamenti nel file system, provenienti dalle directory).

Quando si apre più volte lo stesso file, o comunque ciò che fa capo allo stesso inode, nella tabella di inode si ha sempre solo una voce, dove il contatore dei riferimenti interni serve a sapere quante volte risulta aperto. Quando poi tale contatore arriva a zero, perché i file vengono chiusi mano a mano, la voce della tabella è libera e può essere riutilizzata per un altro inode, oppure può essere semplicemente ripresa così come si trova, se il file viene riaperto (incrementando nuovamente il contatore).



Va però osservato che, durante il funzionamento del sistema operativo, potrebbe farsi riferimento a inode astratti, privi del collegamento a un dispositivo o super blocco. Questo caso riguarda in particolare i condotti «privi di nome», ovvero quelli che non derivano dall'apertura di un file speciale di tipo FIFO.

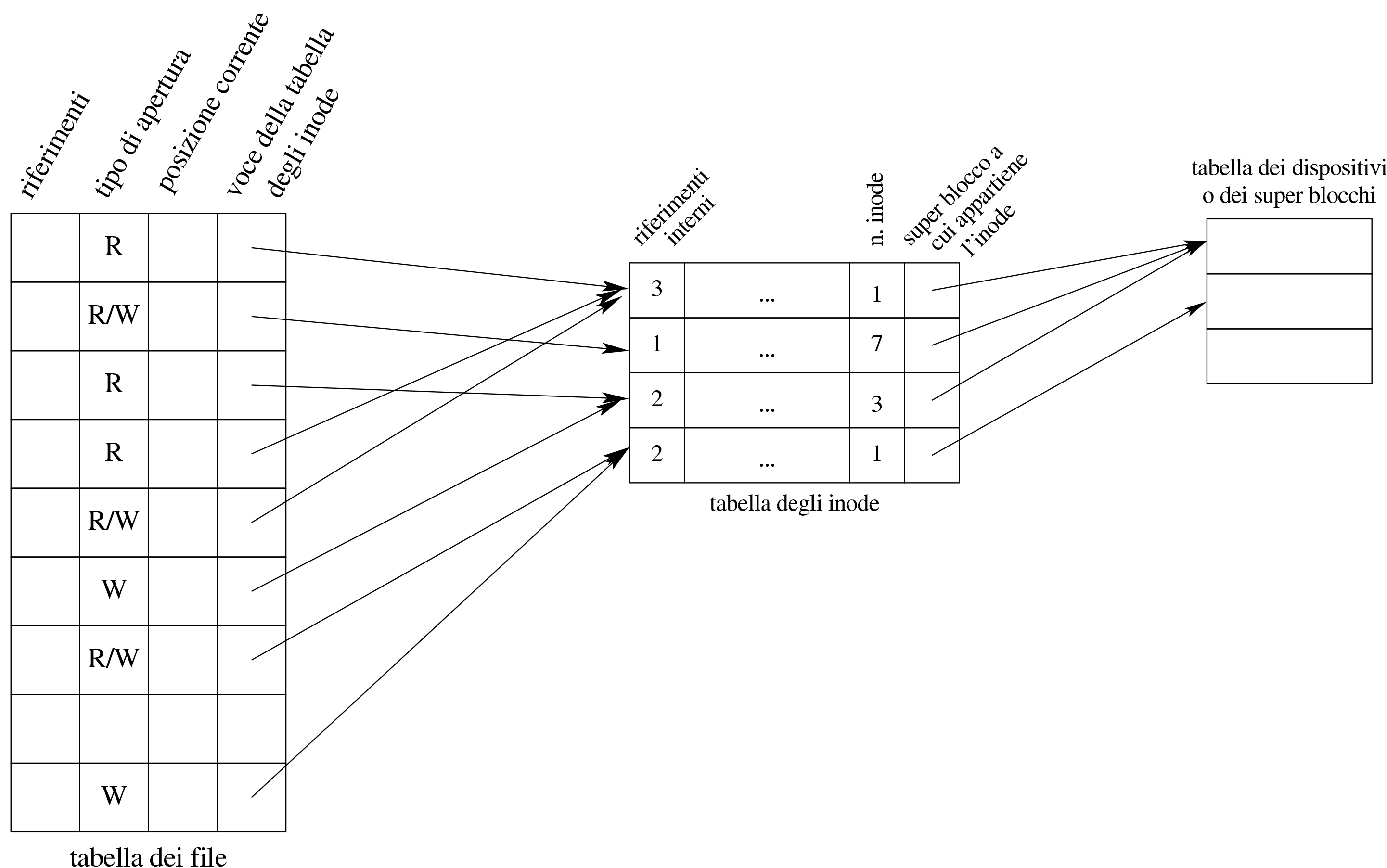
68.6.5.3 Tabella dei file

«

L'apertura di un file, oltre che coinvolgere la tabella degli inode, implica l'aggiunta di una voce nella tabella dei file di sistema (ovvero dei file aperti complessivamente nel sistema operativo). Le voci di questa tabella devono avere un riferimento all'inode, la modalità di apertura (lettura, scrittura o entrambe), la posizione corrente nel file per le letture o le scritture successive, un contatore di riferimenti interni.

L'apertura di un file implica sempre l'aggiunta di una nuova voce nella tabella dei file, ma ci sono delle situazioni in cui uno stesso

processo o più processi differenti possono condividere la stessa voce della tabella dei file di sistema. Come nel caso degli inode, quando il contatore dei riferimenti raggiunge lo zero, significa che la voce corrispondente è chiusa (o libera) e può essere riutilizzata per il prossimo file da aprire.

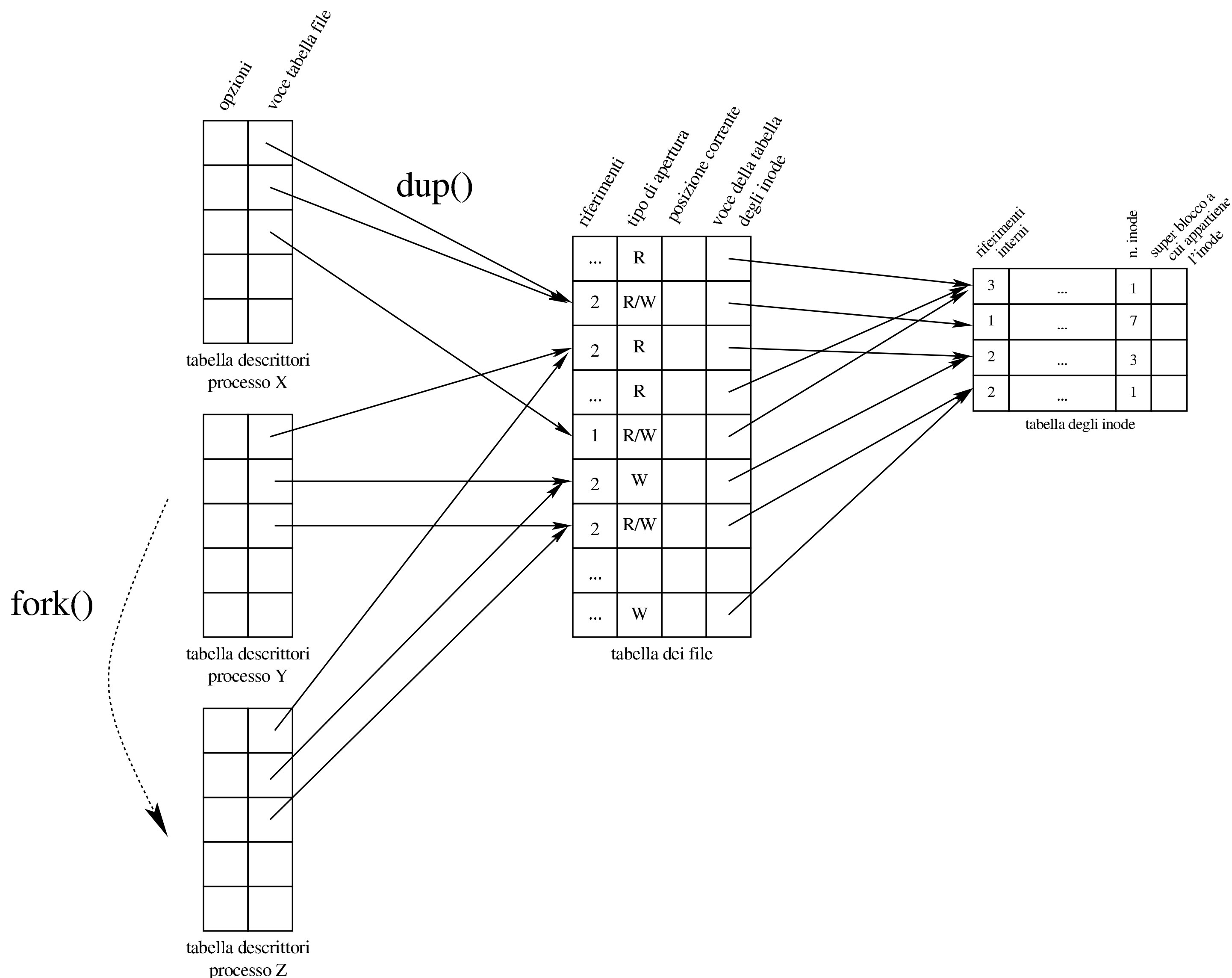


68.6.5.4 Tabella dei descrittori dei file

«

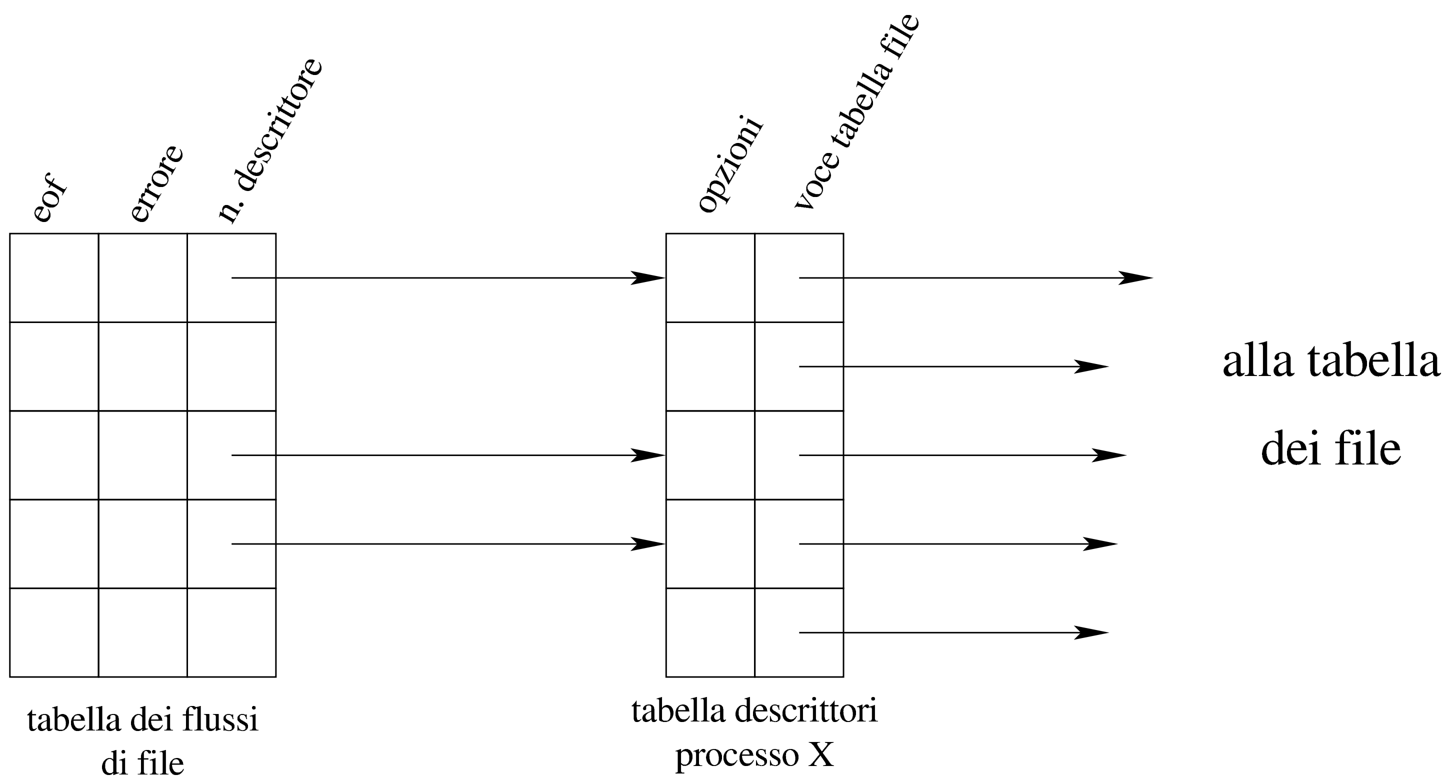
Ogni processo elaborativo ha una propria tabella dei descrittori dei file, nella quale, le voci rappresentano i file aperti dal processo stesso. Le voci della tabella includono il riferimento alla tabella dei file di sistema e le opzioni date in fase di apertura, riguardanti aspetti più precisi rispetto alla semplice distinzione di un accesso in lettura o in scrittura.

Quando un processo elaborativo si sdoppia, attraverso la chiamata di sistema che fa capo alla funzione ***fork()***, i descrittori dei file vengono duplicati e i riferimenti corrispondenti nella tabella dei file di sistema si incrementano. Nello stesso modo, se un processo elaborativo utilizza la funzione della libreria standard ***dup()***, ottiene la duplicazione di un descrittore, incrementando il contatore dei riferimenti nella tabella dei file.



68.6.5.5 Flussi di file

Dal punto di vista del processo elaborativo, la gestione dei file in forma di descrittori o di flussi, potrebbe sembrare indipendente, ma in pratica ciò non può essere. La gestione dei flussi di file implica la presenza di una tabella aggiuntiva (oltre a quella dei descrittori), contenente, per ogni voce, il riferimento al descrittore, un indicatore di errore e un altro indicatore di fine file.



Come suggerisce intuitivamente il disegno, in un sistema operativo POSIX, un flusso di file aperto ha un proprio descrittore di file corrispondente, anche se nell'ambito del programma può rimanere sconosciuto il numero del descrittore abbinato. Tuttavia, per converso, è possibile aprire un file attraverso un descrittore, senza che sia coinvolto necessariamente il flusso che gli corrisponderebbe. A tale proposito, lo standard prescrive la presenza di funzioni che consentono di ristabilire il collegamento esplicito tra flussi e descrittori.

68.7 Il file system Minix 1

Come esempio di come può essere strutturato effettivamente un file system, conforme alle richieste dello standard POSIX, viene proposta la spiegazione dettagliata del tipo usato dal sistema operativo Minix, nelle sue primissime edizioni.

Il kernel Linux consente di accedere a file system Minix 1, eventualmente con l'estensione dei nomi a 30 byte, mentre manca una gestione efficace del file system Minix 2 e manca del tutto la possibilità di accedere alla versione Minix 3.

68.7.1 Blocchi e zone

Il file system Minix 1 suddivide lo spazio disponibile in ***blocchi*** da 1024 byte; così, qualunque oggetto sia memorizzato occupa un multiplo di tale dimensione. Di solito, le unità di memorizzazione di massa sono organizzate in settori da 512 byte, pertanto tale organizzazione in blocchi si adatta perfettamente alle unità comuni.

Per l'indirizzamento dei dati, all'interno del file system, si utilizza il concetto di ***zona***, corrispondente a un multiplo del blocco, ottenuto però come potenza di due. Pertanto possono esserci zone della stessa dimensione dei blocchi, oppure doppie, quadruple,... In pratica, deve essere possibile rappresentare con un numero intero, il logaritmo in base due del rapporto tra la dimensione della zona e la dimensione del blocco.

$$\log_2 \frac{dim_zona}{dim_blocco}$$

Per esempio, una zona da 8192 byte, porta a un rapporto tra zona e blocco di 8 e $log_2 8$ è pari a 3 (in quanto $2^3 = 8$).

Il valore del logaritmo in base due, del rapporto tra zona e blocco, fa parte delle informazioni contenute nel file system Minix 1, perché serve a ottenere la dimensione della zona, attraverso lo scorrimento a sinistra del valore 1024. Per esempio così:

```
y = 1024 << 3;
```

In tal caso, la variabile ***y*** va a contenere il valore 8192. Ma in alternativa, basta calcolare i multipli di blocco, ottenendo così il valore 8:

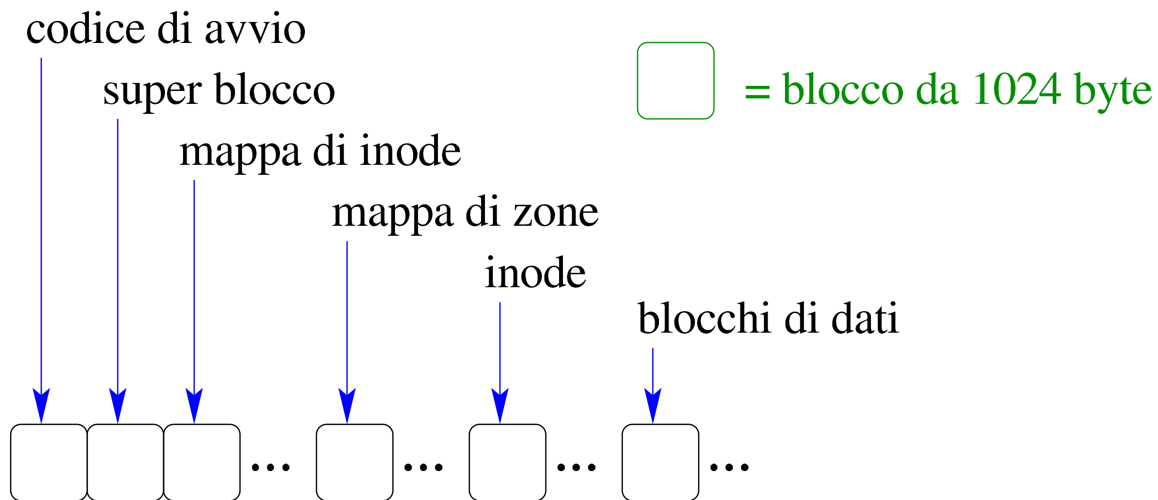
```
y = 1 << 3;
```

Nella tradizione Unix non esiste la «zona», la quale riguarda specificatamente il file system Minix. La zona di Minix rappresenta in pratica il concetto di «blocco» dei sistemi Unix.

68.7.2 Struttura generale

«

La struttura generale del file system Minix 1 è schematizzata dalla figura successiva. Il primo blocco (da 1024 byte) è riservato al codice di avvio, anche se di solito questo occupa soltanto un settore da 512 byte. Il secondo blocco contiene il «super blocco», ovvero l'intestazione del file system, con le informazioni generali sullo stesso. Il terzo blocco, ed eventuali altri blocchi successivi, sono utilizzati per una mappa degli inode, la quale ha lo scopo di annotare quali sono utilizzati e quali sono liberi. A partire dal blocco successivo inizia la mappa delle zone utilizzate (zone, intese come multipli dei blocchi, come spiegato nella sezione precedente). Dopo la mappa delle zone appaiono i blocchi contenenti gli inode (tanti quanti sono previsti nella mappa di inode). Successivamente appaiono i blocchi usati dalle zone di dati che utilizzano lo spazio rimanente.



Va osservato che se le zone hanno una dimensione maggiore dei blocchi, il primo blocco utile per la memorizzazione dei dati (dopo gli inode) deve iniziare all'inizio di una zona, contando le zone a partire dal primo blocco (quello riservato dal codice di avvio). Pertanto, potrebbero rimanere anche blocchi non utilizzabili, dopo quelli delle mappe e prima di quelli dei dati.

68.7.3 Super blocco

«

Il super blocco raccoglie le informazioni più importanti del file system e dalla sua integrità dipende l'accessibilità di tutto il resto del contenuto presente. Anche se gli viene riservato un blocco intero, in pratica, il super blocco di Minix 1 occupa molto meno spazio.

Il secondo campo del super blocco, come si vede dalla figura, rappresenta la dimensione dell'unità di memorizzazione (o della partizione considerata), espressa in zone. Per esempio, se si utilizzano zone uguali ai blocchi, un dischetto da 1 440 Kibyte è composto esattamente da 1 440 zone.

Il quinto campo indica la prima zona dati, contando a partire da zero. La prima zona dati è la prima zona che possa essere usata, dopo gli inode. Nella mappa delle zone utilizzate, il bit che rappresenta la zona dati numero uno, si riferisce a questa prima zona dati (nella mappa delle zone, la zona dati zero risulta sempre utilizzata ma in realtà non esiste).

Il sesto campo indica il logaritmo in base due, del rapporto tra di-

mensione della zona e del blocco. Per esempio, un valore pari a zero indica che la zona è uguale al blocco; uno indica che la zona è costituita da due blocchi; tre indica che la zona è composta da quattro blocchi e così di seguito.

Il «numero magico» è il codice di riconoscimento, usato per verificare che si tratti effettivamente di un file system Minix 1. Tale numero deve essere $137F_{16}$.

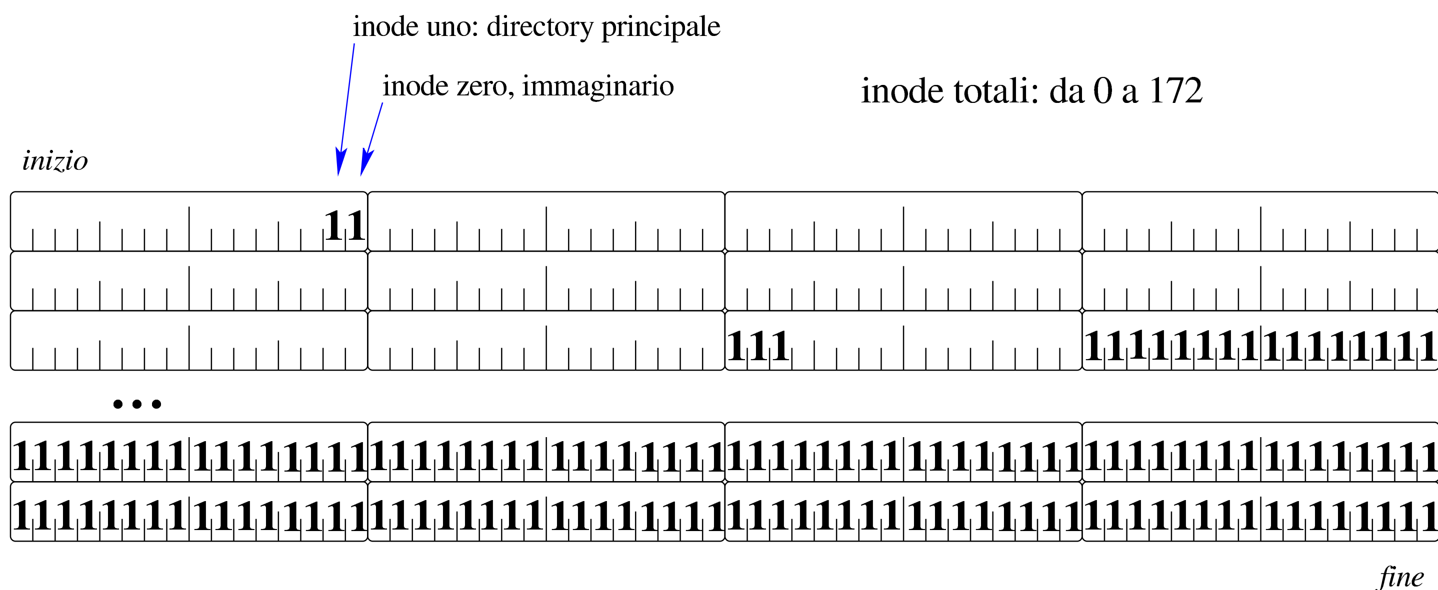
68.7.4 Mappa di inode

«

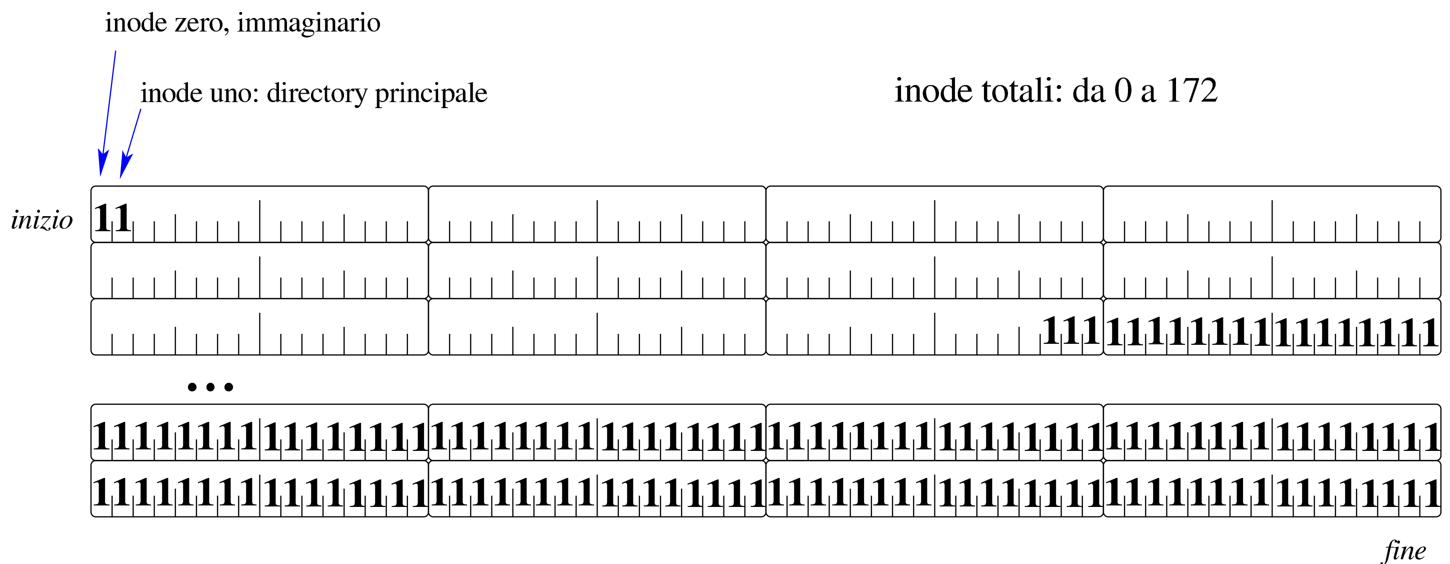
La mappa di inode è costituita da un insieme di bit, ognuno dei quali rappresenta lo stato di utilizzazione di un certo inode: 1 indica un inode utilizzato; 0 indica un inode libero. In questa mappa, il primo bit, riferito all'inode zero, è sempre attivo. Ma l'inode zero, in pratica, non viene rappresentato e il primo vero inode, ovvero quello riferito alla directory radice, ha sempre il numero uno.

In base al fatto che l'inode zero, in pratica, non esiste, anche se risulta sempre utilizzato, va considerato che il valore presente nel primo campo del super blocco indica la quantità reale di inode, ovvero l'indice massimo (partendo da zero) che si può utilizzare nella loro scansione.

La mappa di bit va però scandita, suddividendola a blocchi da 16 bit. L'esempio seguente rappresenta una mappa per 172 inode, dove si vede il primo (zero) impegnato e il secondo che già è predisposto per la directory principale:



Nel disegno sono rappresentati solo i bit a uno, lasciando gli altri come spazi vuoti. Va osservato che l'ordine in cui si dispongono i bit non è quello che ci si aspetterebbe: il primo, quello dell'inode zero, appare a destra del suo insieme di 16 bit (si tratta quindi del bit meno significativo), mentre lo si attenderebbe a sinistra secondo il senso di lettura latino. Considerato che i bit inutilizzati vanno posti a uno, come se esistessero altrettanti inode impegnati, l'ultimo insieme di 16 bit va interpretato con attenzione. Per avere una visione più umana della mappa, occorrerebbe invertire la sequenza di bit in ogni gruppetto:



Al problema dell'inversione della sequenza di bit, si aggiunge il fatto che il file system è nato per un'architettura *little endian*, ovvero a byte invertiti, ma la questione viene trattata alla fine del capitolo.

Per sapere dove si trova un certo inode ***n***, occorre considerare che questi si collocano dopo i blocchi della mappa di zone, che il primo vero inode è quello con indice uno, ovvero il secondo, in base alla numerazione della mappa. Come viene descritto successivamente, ogni inode occupa 32 byte, pertanto, in ogni blocco ci stanno esattamente 32 inode.

68.7.5 Mappa di zone

«

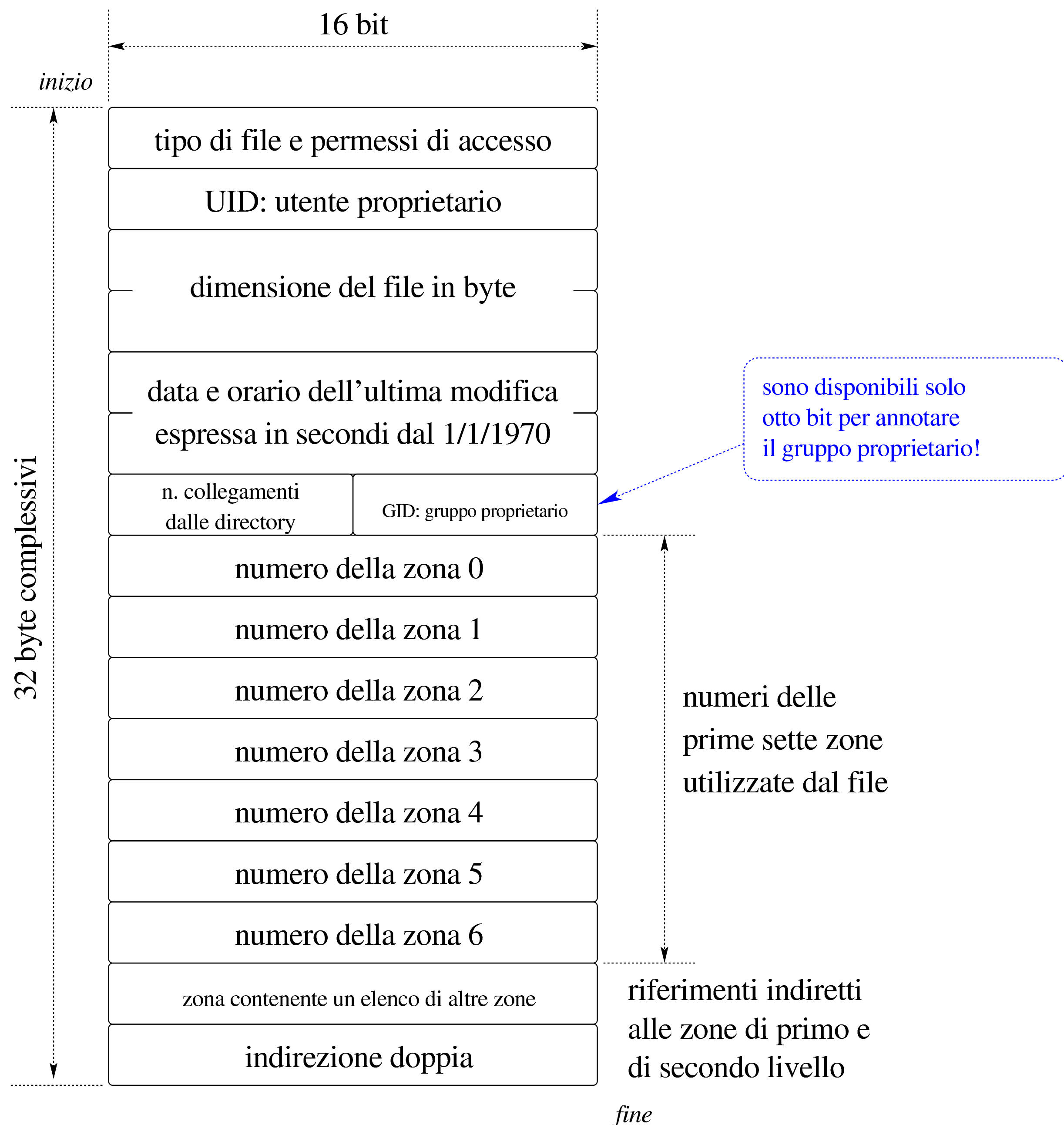
La mappa delle zone dei dati, funziona come quella di inode, dove i bit a uno indicano una zona utilizzata e il primo bit, riferito alla zona zero, è sempre a uno, ma in realtà la prima vera zona dati è quella a cui corrisponde l'indice uno. Come per la mappa di inode, anche in questo caso valgono le stesse considerazioni relative al fatto che la scansione deve essere fatta a gruppi di 16 bit e che il conteggio inizia dalla parte numericamente meno significativa di tali gruppi.

È bene precisare che la mappa si riferisce alle zone dei dati, pertanto riguarda quelle zone che iniziano dopo tutte le informazioni già descritte, compresa la stessa mappa e la tabella di inode successiva. Per fare un esempio, se nel super blocco è scritto che la prima zona dati è quella con il numero 19, significa che il bit con indice uno della mappa (il secondo) individua la zona 19 e le zone precedenti non possono essere utilizzate per i dati.

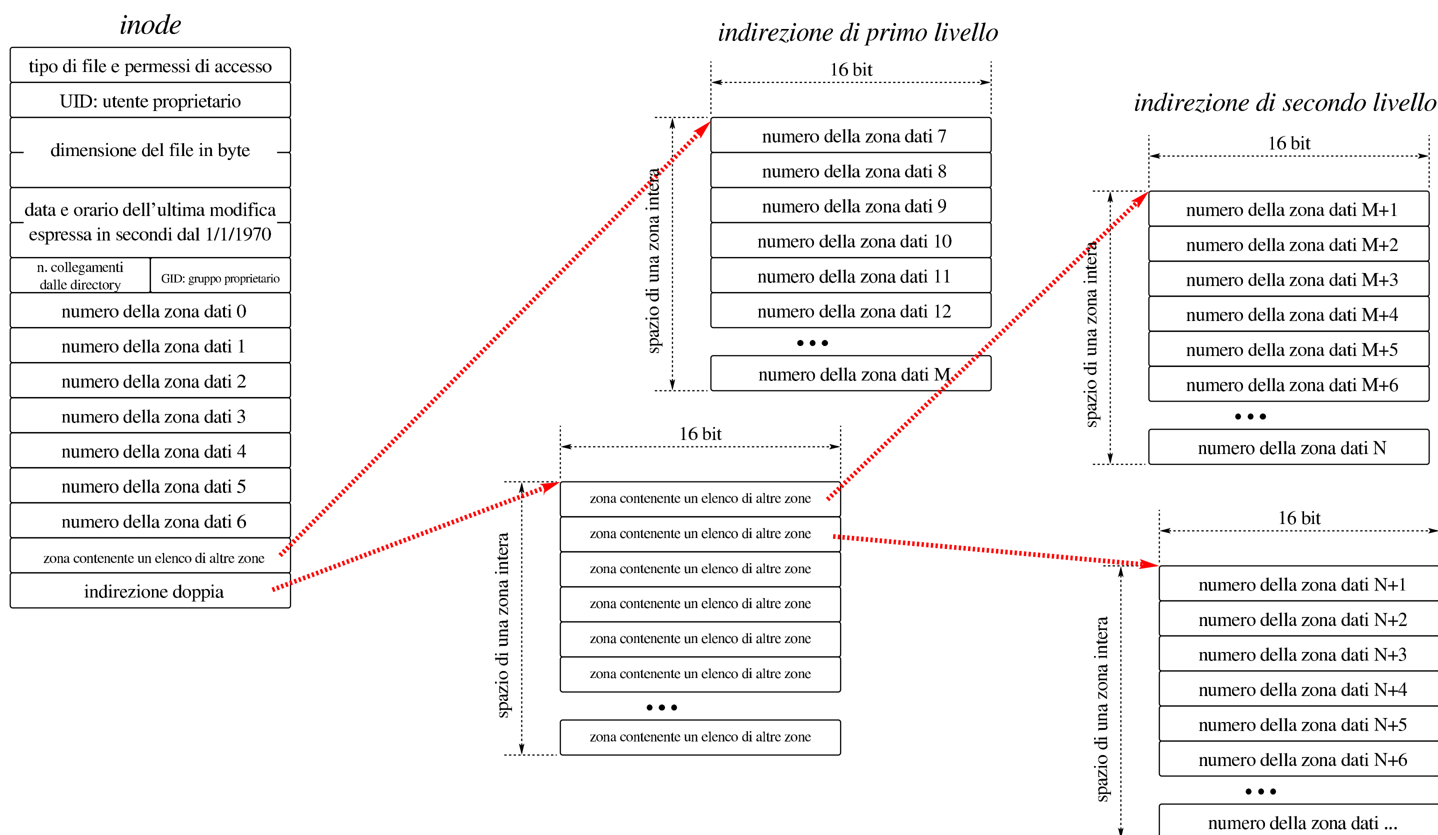
68.7.6 Inode

«

I blocchi successivi alla mappa delle zone dei dati, sono utilizzati per gli inode, di cui si conosce la quantità, perché questa è annotata nel super blocco. Nel file system Minix 1, ogni inode occupa esattamente 32 byte.



La figura successiva mostra il meccanismo usato per indirizzare file che occupano più di sette zone, attraverso elenchi aggiuntivi, ognuno dei quali occupa a sua volta una zona intera. I riferimenti indiretti alle zone possono essere quindi di primo livello, o di secondo livello, come suggerito dalla figura stessa.

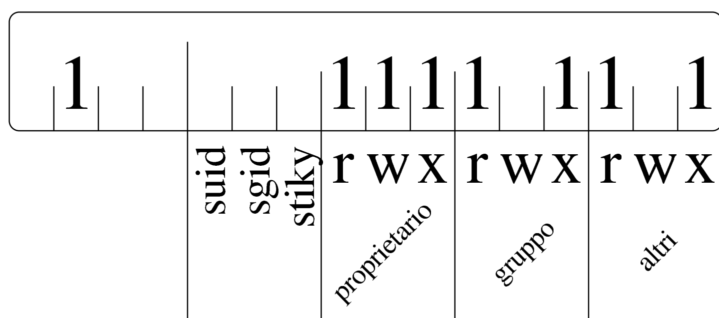


Ipotizzando di avere zone della stessa dimensione dei blocchi (1024 byte), dal momento che gli elenchi indiretti possono contenere a loro volta 512 numeri di zona, sarebbe possibile gestire file con una dimensione massima di 7+512+512×512 Kibyte, ovvero 262663 Kibyte. Disponendo di zone della dimensione di quattro blocchi, si potrebbero gestire file da 7+2048+2048×2048 Kibyte, ovvero 4196359 Kibyte. Con lo stesso criterio, con zone da otto blocchi, si potrebbero gestire file da poco più di 16 Gibyte; con zone da 16 blocchi si arriverebbe a poco più di 64 Gibyte. A questi limiti si aggiunge però il fatto che le zone sono individuate da numeri a 16 bit; pertanto, con zone da un solo blocco, si possono indirizzare al massimo 65536 Kibyte, ovvero 64 Mibyte; con zone da due blocchi si arriva a 128 Mibyte; con zone da 16 blocchi si arriva al massimo a 1 Gibyte. Pertanto, la doppia «indirezione» può essere usata solo parzialmente e non avrebbe senso un'indirezione tripla.

A parte la limitazione nella dimensione dei file, va annotato un fatto che può risultare più spiacevole: il numero del gruppo proprietario del file (GID) viene rappresentato con soli 8 bit. Ciò significa che si possono indicare gruppi fino al numero 255 e in pratica, quando vi si copia un file, il numero del gruppo viene troncato nella parte più significativa. Un altro limite importante riguarda il fatto che l'inode riporti solo la data di modifica del file, mancando così la data di accesso e la data di creazione dell'inode stesso.

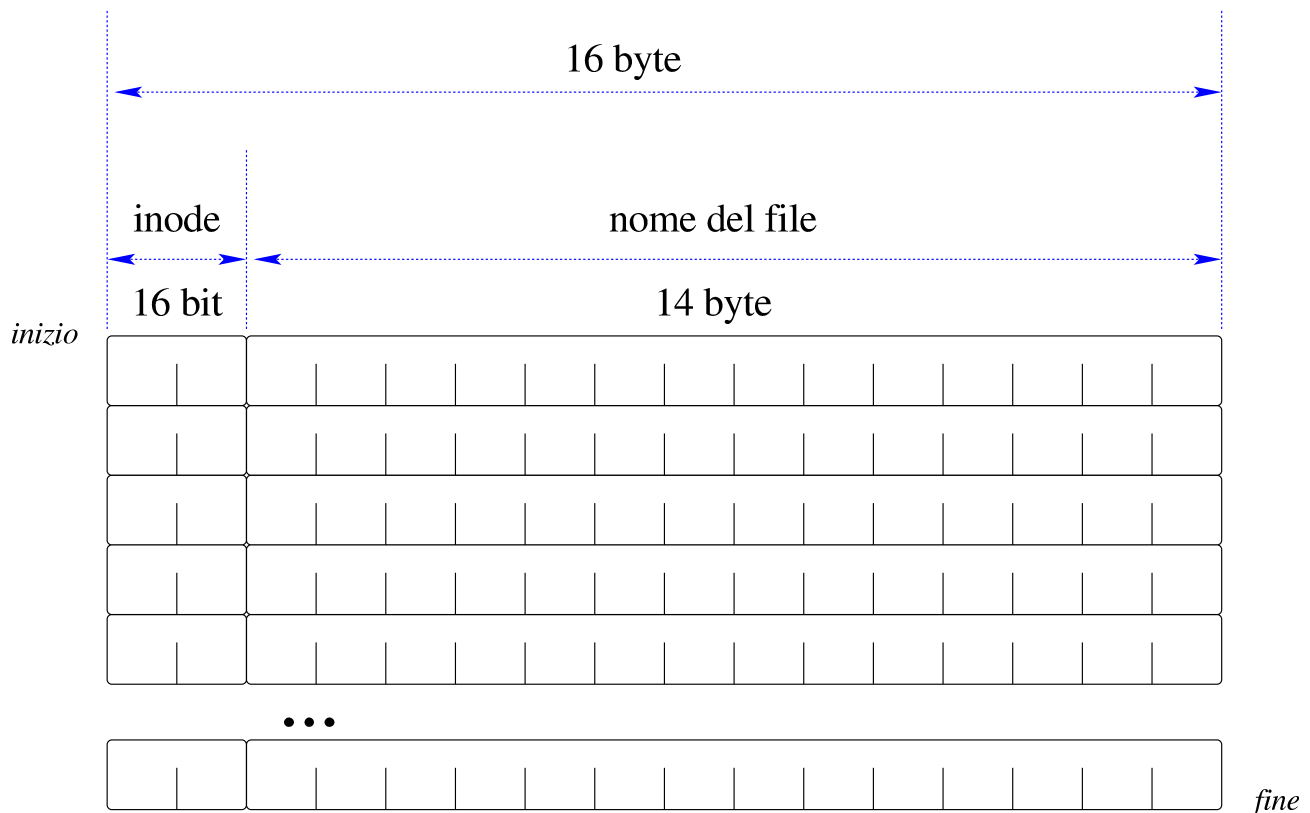
Il primo campo da 16 bit di un inode, rappresenta il tipo e i permessi del file a cui si riferisce l'inode (si veda anche la sezione 70.2.1 a proposito della «modalità» POSIX). L'interpretazione di questo valore deve avvenire secondo gli standard dei sistemi POSIX, ovvero secondo lo schema seguente, dove si ipotizza una directory con permessi di accesso e di lettura per tutti gli utenti:

0 0 0 1	file FIFO
0 0 1 0	dispositivo a caratteri
0 1 0 0	directory
0 1 1 0	dispositivo a blocchi
1 0 0 0	file normale
1 0 1 0	collegamento simbolico
1 1 0 0	socket di dominio Unix



68.7.7 Directory

Le directory si collocano nelle zone dei dati, come gli altri file, e a loro si accede a partire da un inode (come per tutti gli altri file). La struttura di una directory è molto semplice, come si può vedere nella figura successiva:



In pratica, si tratta di un file suddiviso in *record* a dimensione fissa da 16 byte, dove i primi due byte rappresentano il numero inode del nome che occupa i restanti 14 byte. Il file system Minix 1 consente così di rappresentare nomi di file fino a un massimo di 14 caratteri, ma nei sistemi GNU/Linux si utilizza spesso un'estensione con directory aventi *record* da 32 byte, in modo da poter rappresentare nomi fino a 30 caratteri.

La directory è un file come gli altri, pertanto, per sapere quante sono le voci che la compongono, occorre conoscere la dimensione del file, come annotato nel suo inode. Per esempio, una directory con quattro voci (inclusi i nomi '`.`' e '`..`'), occupa 64 byte.

68.7.8 Il problema dell'inversione dei byte

«

Il sistema operativo Minix nasce negli anni 1980 per elaboratori a 16 bit con l'inversione dei byte (*little endian*). Per questa ragione, i byte che costituiscono l'organizzazione del file system Minix 1 sono invertiti. Ciò diventa un problema quando si legge il contenuto del file system in modo diretto, in esadecimale, perché tutte le voci che prevedono una rappresentazione a 16 o a 32 bit, vanno rovesciate in modo appropriato, per poterle interpretare correttamente. Per esempio, in un'altra sezione è stato descritto il modo in cui viene popolata la mappa degli inode e delle zone dei dati; ai problemi lì descritti si aggiungerebbe anche l'inversione dei byte.

Se si considera che un file system serve per scrivere dati anche su unità di memorizzazione rimovibili, utilizzabili presumibilmente su altri sistemi e altre architetture, sarebbe più appropriata una progettazione che preveda sempre la scrittura «ordinata» dei byte, come si fa per i dati trasmessi in rete. Nel caso particolare di Minix, con l'evolvere del sistema e con l'adattamento anche ad altre architetture, si è reso necessario considerare se il file system a cui si accede giunge ordinato secondo la propria architettura, oppure se per questa è inverso. In pratica, un numero magico $137F_{16}$ indica che il file system va bene così; altrimenti, il numero $7F13_{16}$ richiede che i valori a 16 e a 32 siano invertiti (byte per byte), per poter essere interpretati correttamente.

68.8 Creazione ed eliminazione di file di qualunque tipo

«

Un file «normale», definito in inglese come *regular file*, è il contenitore di una sequenza di byte, rappresentato in qualche modo nel

file system. Nei sistemi Unix, anche le directory sono dei file, benché si tratti evidentemente di un tipo speciale, per il quale si richiede un trattamento particolareggiato; inoltre, altri tipi di entità rientrano nella gestione complessiva del concetto di file per i sistemi Unix. Originariamente è stato usato il termine «nodo», da cui deriva il nome della funzione ***mknod()***, con cui si poteva creare qualunque tipo di file.

68.8.1 La funzione «mknod()»

La funzione ***mknod()***, dichiarata nel file di intestazione '`sys/stat.h`', potenzialmente, è in grado di creare qualunque tipo di file, ma completamente vuoto, ammesso che si tratti di un tipo di file che ha un contenuto rappresentato nel file system. Teoricamente questa funzione potrebbe creare anche delle directory, ammesso che il sistema operativo lo consenta, ma si tratterebbe comunque di directory prive delle voci obbligatorie '`.`' e '`..`', quindi si tratterebbe di directory incomplete ed errate per il file system.

```
int mknod (const char *path, mode_t mode, dev_t dev);
```

Il primo parametro della funzione è una stringa che rappresenta il percorso del file da creare nel file system; il secondo parametro, ***mode***, individua il tipo di file ed eventualmente i permessi di accesso; l'ultimo parametro, ***dev***, il numero del file di dispositivo, ammesso che si tratti della creazione di questo tipo di file. La tabella successiva elenca le macro-variabili da usare per comporre il valore del parametro ***mode***, usando l'operatore OR binario, avendo la cura di specificare una sola macro-variabile per il tipo.

Tabella 68.76. Macro-variabili per esprimere, complessivamente il tipo e i permessi di un file.

Macro-variabile	Valore numerico equivalente	Significato
`S_IFBLK`	vedere ‘`sys/stat.h`’	File di dispositivo a blocchi.
`S_IFCHR`	vedere ‘`sys/stat.h`’	File di dispositivo a caratteri.
`S_IFIFO`	vedere ‘`sys/stat.h`’	File FIFO.
`S_IFREG`	vedere ‘`sys/stat.h`’	File normale (*regular file*).
`S_IFDIR`	vedere ‘`sys/stat.h`’	Directory.
`S_IFLNK`	vedere ‘`sys/stat.h`’	Collegamento simbolico.
`S_IFSOCK`	vedere ‘`sys/stat.h`’	Socket di dominio Unix.
`S_ISUID`	4000_8	Rappresenta l’attivazione del bit S-UID.
`S_ISGID`	2000_8	Rappresenta l’attivazione del bit S-GID.
`S_ISVTX`	1000_8	Rappresenta l’attivazione del bit Sticky.
`S_IRWXU`	0700_8	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per l’utente proprietario.
`S_IRUSR`	0400_8	Rappresenta il permesso di lettura per l’utente proprietario.
`S_IWUSR`	0200_8	Rappresenta il permesso di scrittura per l’utente proprietario.

Macro-variabile	Valore numerico equivalente	Significato
S_IXUSR	0100_8	Rappresenta il permesso di esecuzione o attraversamento per l'utente proprietario.
S_IRWXG	0070_8	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per il gruppo proprietario.
S_IRGRP	0040_8	Rappresenta il permesso di lettura per il gruppo proprietario.
S_IWGRP	0020_8	Rappresenta il permesso di scrittura per il gruppo proprietario.
S_IXGRP	0010_8	Rappresenta il permesso di esecuzione o attraversamento per il gruppo proprietario.
S_IRWXO	0007_8	Rappresenta tutti i permessi di lettura, scrittura ed esecuzione o accesso, per gli altri utenti.
S_IROTH	0004_8	Rappresenta il permesso di lettura per gli altri utenti.
S_IWOTH	0002_8	Rappresenta il permesso di scrittura per gli altri utenti.
S_IXOTH	0001_8	Rappresenta il permesso di esecuzione o attraversamento per gli altri utenti.

68.8.1.1 Creazione di un file «normale»

«

L'esempio seguente mostra l'uso della funzione ***mknod()*** per la creazione di un file comune:

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mknod ("/tmp/miofile",
                  (mode_t) (S_IFREG | S_IRUSR | S_IWUSR
                            | S_IXUSR),
                  (dev_t) 0);
  if (status != 0) perror (NULL);
  return (0);
}
```

68.8.1.2 Creazione di una directory

L’esempio seguente mostra un programma elementare che ha lo scopo di creare una directory vuota, priva anche delle voci obbligatorie ‘`.`’ e ‘`..`’. In condizioni normali, il sistema operativo dovrebbe impedire tale azione, producendo un messaggio di errore.

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mknod ("/tmp/miadir",
                  (mode_t) (S_IFDIR | S_IRUSR | S_IWUSR
                            | S_IXUSR),
                  (dev_t) 0);
  if (status != 0) perror (NULL);
  return (0);
}
```

68.8.1.3 Creazione di un file FIFO

«

Un file FIFO è un «condotto» (*pipe*) rappresentato da un file e si distingue dai condotti creati internamente, senza tale associazione simbolica. Pertanto si distingue anche tra «condotti con nome» (*pipe* con nome) o file FIFO e «condotti senza nome» (*pipe* senza nome) o solo *pipe*.

L'esempio seguente mostra un programma elementare che ha lo scopo di creare un file FIFO.

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mknod ("/tmp/miofifo",
                  (mode_t) (S_IFIFO | S_IRUSR | S_IWUSR
                            | S_IXUSR),
                  (dev_t) 0);
  if (status != 0) perror (NULL);
  return (0);
}
```

68.8.1.4 Creazione di file di dispositivo

La creazione di un file di dispositivo richiede l'indicazione del numero del dispositivo. Ciò comporta una complicazione, perché quel numero contiene simultaneamente le informazioni sul numero primario e sul numero secondario.

Originariamente, il numero del dispositivo era formato da 16 bit, di cui l'ottetto (il byte) più significativo rappresentava il numero primario, mentre quello meno significativo il numero secondario. Dal momento che questa organizzazione è sì quella tradizionale, ma non è richiesta dallo standard, diventa necessario disporre di funzioni o macroistruzioni che aiutino a comporre correttamente il numero di dispositivo complessivo, o a estrapolare le sue componenti.

Vari sistemi che si rifanno al modello di Unix introducono tre funzioni o macroistruzioni, utili per manipolare i numeri di dispositivo. Purtroppo queste funzioni non sono standard, benché abbastanza

diffuse:

```
dev_t makedev (int major, int minor);
```

```
int   major   (dev_t device);
```

```
int   minor   (dev_t device);
```

La funzione ***makedev()*** assembla il numero primario e il numero secondario ottenuti come argomenti, restituendo un numero di dispositivo complessivo; per converso, le funzioni ***major()*** e ***minor()*** estrapolano rispettivamente il numero primario e il numero secondario, a partire da un numero di dispositivo complessivo. Tali funzioni dovrebbero essere dichiarate nel file di intestazione '`sys/types.h`'.

I due esempi seguenti mostrano la creazione di due file di dispositivo, uno a caratteri e uno a blocchi. Ci si avvale della funzione ***makedev()*** per assemblare il numero di dispositivo complessivo, a partire dal numero primario e dal numero secondario.

```
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mknod ("/tmp/miodev1",
                  (mode_t) (S_IFCHR | S_IRUSR | S_IWUSR
                           | S_IXUSR),
                  makedev (1, 2));
  if (status != 0) perror (NULL);
  return (0);
}
```

```
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mknod ("/tmp/miodev2",
                  (mode_t) (S_IFBLK | S_IRUSR | S_IWUSR
                           | S_IXUSR),
                  makedev (3, 4));
  if (status != 0) perror (NULL);
  return (0);
}
```

Va osservato che la creazione di un file di dispositivo dovrebbe risultare concessa solo a un processo in funzione con i privilegi dell'utente con numero UID pari a zero ('**`root`**').

68.8.2 La funzione «mkdir()»

«

La funzione ***mkdir()*** costituisce il modo corretto per creare una directory vuota (ma provvista delle voci '`.`' e '`..`' obbligatorie).

```
int mkdir (const char *path, mode_t mode);
```

A differenza di ***mknod()***, il parametro ***mode*** va usato esclusivamente per indicare i permessi di accesso richiesti, tenendo conto, naturalmente, che questi vengono filtrati ulteriormente in base alla maschera dei permessi (*user mask*). In altri termini, nel parametro ***mode*** non si può specificare il tipo di file, cosa che comunque sarebbe ignorata, dato che si tratta della creazione di una directory e di nulla altro.

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mkdir ("/tmp/miadir",
                  (mode_t) (S_IRUSR | S_IWUSR | S_IXUSR));
  if (status != 0) perror (NULL);
  return (0);
}
```

L'esempio mostra la creazione della directory '`/tmp/miadir/`' in un programma completo e molto semplice.

68.8.3 La funzione «mkfifo()»

«

La funzione ***mkfifo()*** consente di creare un file FIFO, specificando il percorso e i permessi, in modo analogo a quanto si farebbe con ***mkdir()*** per la creazione delle directory, con la differenza che in questo caso l'uso della funzione ***mknod()*** sarebbe comunque corretto.

```
int mkfifo (const char *path, mode_t mode);
```

Segue un esempio molto semplice, al pari di quelli già apparsi nel capitolo.

```
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = mkfifo ("/tmp/miofifo",
                   (mode_t) (S_IRUSR | S_IWUSR | S_IXUSR));
  if (status != 0) perror (NULL);
  return (0);
}
```

68.8.4 La funzione «unlink()»

«

La funzione ***unlink()*** consente di eliminare un file, possibilmente di qualunque tipo, «scollegandolo» dalla directory a cui si riferisce il percorso indicato per l'operazione. Dal momento che un file è rappresentato in un file system Unix da un inode, tale inode viene eliminato effettivamente se non ci più altri riferimenti allo stesso.

In linea di principio, con ***unlink()*** non dovrebbe essere possibile la cancellazione di una directory; inoltre, se nel frattempo il file in questione risulta utilizzato da un processo, l'operazione di cancellazione (scollegamento) dovrebbe completarsi soltanto nel momento in cui il file risulta chiuso a tutti gli effetti.

```
int unlink (const char *path);
```

L'esempio seguente mostra la cancellazione del file '`/tmp/cancellami`':

```
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = unlink ("/tmp/cancellami");
  if (status != 0) perror (NULL);
  return (0);
}
```

68.8.5 La funzione «rmdir()»

«

La funzione ***rmdir()*** consente di eliminare una directory, purché vuota (contenente soltanto le voci '`.`' e '`..`'), specificandone il percorso.

```
int rmdir (const char *path);
```

L'esempio seguente mostra la cancellazione della directory '`/tmp/cancellami/`':

```
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
int
main (void)
{
  int status;
  status = rmdir ("/tmp/cancellami");
  if (status != 0) perror (NULL);
  return (0);
}
```

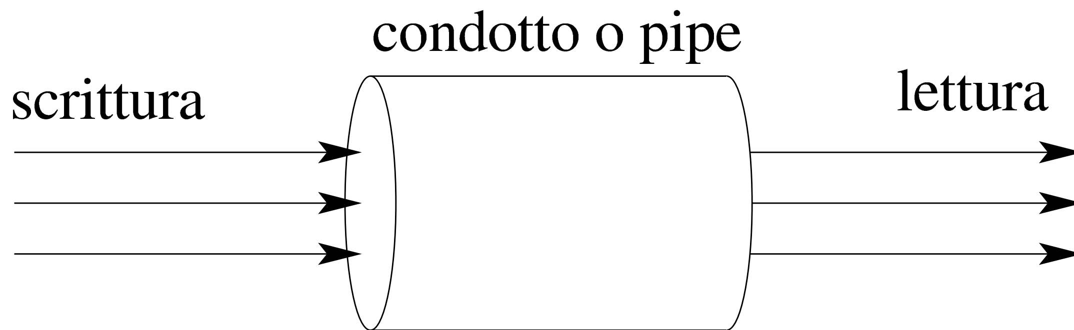
68.8.6 La funzione «remove()»

La funzione ***remove()*** cancella un file, utilizzando ***unlink()*** oppure ***rmdir()***, in base al tipo di file specificato per la rimozione stessa.

```
int remove (const char *path);
```

68.9 Condotti

I ***condotti***, o *pipe*, sono dei file virtuali, ad accesso FIFO (*First in, first out*). In altri termini sono delle ***code***, in forma di file. Un condotto richiede che ci siano sia processi che vi scrivono, sia processi che vi leggono le informazioni. La lettura comporta il prelievo di dati e la liberazione di spazio disponibile per ulteriori operazioni di scrittura.

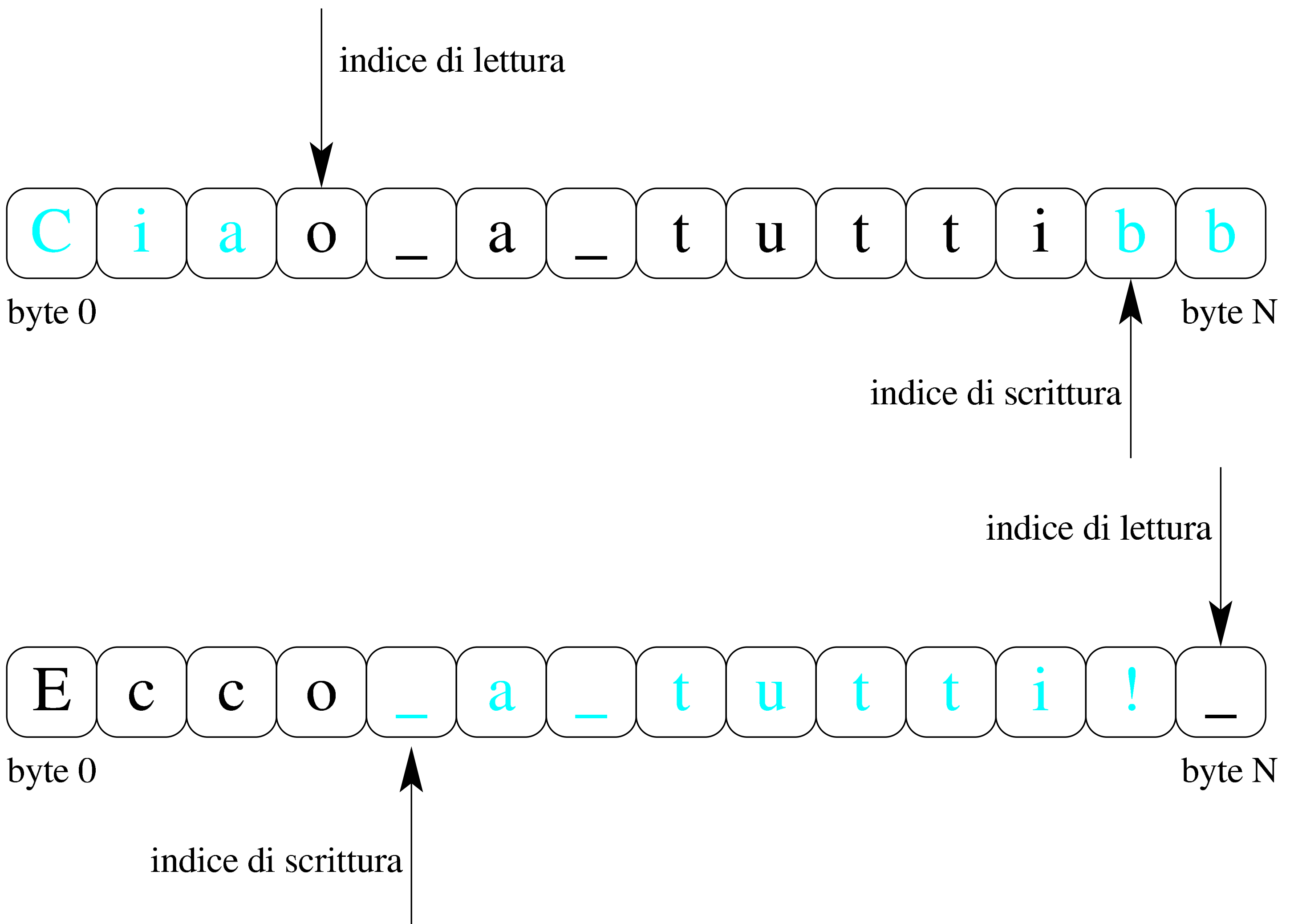


Se più di un processo apre uno stesso condotto in scrittura, non è possibile prevedere l'ordine in cui le operazioni di scrittura procedono; nello stesso modo, se più processi aprono uno stesso condotto in lettura, non è possibile prevedere con quale ordine vengano raccolti i dati dal condotto stesso. In altri termini, è compito dei processi di definire un protocollo tra di loro, se i dati devono confluire secondo un certo criterio.

68.9.1 Realizzazione materiale del condotto

«

In memoria centrale, il condotto si realizza come un array di byte, a cui si accede attraverso due indici: uno per la scrittura e l'altro per la lettura, tenendo conto che raggiunta la fine, si riprende dall'inizio.



Lo schema mostra un array scandito con due indici, dove i caratteri di colore nero rappresentano i byte scritti e ancora da leggere, mentre quelli in azzurro-ciano (ovvero quelli di colore più chiaro), rappresentano i byte già letti che possono essere sovrascritti. Lo schema mostra due momenti differenti, dove nel secondo caso l'indice di scrittura, una volta raggiunta la fine dell'array, riprende dall'inizio.

Dal momento che un condotto viene rappresentato in memoria come un inode, con tanto di elenco di riferimenti ai blocchi utilizzati nel file system, dato che tali annotazioni non servono perché nulla viene memorizzato in un file system, originariamente si utilizzava proprio quella porzione di memoria (quella dei blocchi diretti) per

l'array che consente di conservare temporaneamente i dati. Tuttavia, dal momento che lo standard di oggi richiede che lo spazio nella coda di un condotto sia abbastanza grande, è improbabile che si utilizzi ancora questo metodo.

68.9.2 Condotti «senza nome» e condotti «con nome»

«

Il condotto, come concetto, è un file virtuale già «aperto» e utilizzato da qualche processo elaborativo. In questi termini, un condotto potrebbe essere creato al volo, da un processo che successivamente ne avvia un altro con il quale deve comunicare. Un condotto realizzato al volo non ha alcun riferimento nel file system, pertanto gli si attribuisce la caratteristica di essere «senza nome». D'altro canto, un condotto può essere rappresentato nel file system da un file speciale, di tipo FIFO, da trattare come se fosse un file normale, benché non lo sia. Nel secondo caso si tratta di un condotto «con nome», perché c'è un nome nel file system, ovvero si tratta di un file FIFO.

68.9.3 Protocollo di accesso ai condotti

«

Una volta creato un condotto (che questo sia senza nome o che derivi dall'apertura di un file FIFO, ciò non fa differenza), solo dopo che questo risulta utilizzato sia in scrittura, sia in lettura, si può procedere con le operazioni di scrittura e lettura.

Quando un processo tenta di leggere da un condotto nel quale non sono disponibili dati nuovi, questo viene sospeso, in attesa di dati; nello stesso modo, un processo che tenta di scrivere in un condotto che non ha spazio disponibile (perché i dati già inseriti non sono ancora stati letti), viene sospeso in attesa di tale disponibilità.

Quando un processo tenta di leggere da un condotto che non viene più utilizzato in scrittura (perché non ci sono più descrittori di file associati al condotto in scrittura), si trova di fronte a un file concluso (nel senso che si avvera la condizione di fine del file); quando invece un processo tenta di scrivere in un condotto a cui non corrisponde più alcun descrittore in lettura, questo processo riceve il segnale SIGPIPE e, se il processo lo ignora, l'operazione di scrittura termina con un errore '**EPIPE**'.

Dal momento che la natura di un condotto è quella di essere ad accesso sequenziale, non è possibile posizionare l'indice di lettura o scrittura, con l'ausilio della funzione ***lseek()***.

68.9.4 Funzione «pipe()»

«

La funzione ***pipe()*** crea un condotto al volo, restituendo attraverso un array di due elementi i numeri dei descrittori per l'accesso in lettura e in scrittura. Tale condotto non ha un file FIFO corrispondente, ma risulta ugualmente aperto e disponibile al processo che lo crea.

```
int pipe (int fd[2]);
```

La funzione restituisce l'esito dell'operazione: zero in caso di successo, oppure −1 in caso di problemi (aggiornando il valore della variabile ***errno***). I descrittori per l'accesso vengono ottenuti dal contenuto dell'array ***fd[]***, dove ***fd[0]*** è il descrittore per l'accesso in lettura, mentre ***fd[1]*** è quello per l'accesso in scrittura.

Con questa funzione, un processo crea un condotto e ottiene i due descrittori di accesso; tuttavia, lo scopo di un condotto è quello di mettere in comunicazione due o più processi. Pertanto, dopo un'o-

perazione di questo tipo, si passa quasi certamente a una biforcazione ed eventualmente all'esecuzione di un altro programma. Dopo la biforcazione è normale che il processo originario chiuda il descrittore che non gli serve (dal momento che utilizza probabilmente solo quello di scrittura o solo quello di lettura) e che così faccia anche il processo sdoppiato: il programma che eventualmente venisse caricato al posto del processo sdoppiato troverebbe già tutto pronto per iniziare a lavorare correttamente. Il listato successivo mostra un esempio, disponibile eventualmente presso *allegati/c/esempio-posix-pipe.c* , ottenuto modificando un esempio analogo che appare nella pagina di manuale *pipe(2)* di un sistema GNU/Linux.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//----------------------------------------------------------
int
main (void)
{
  int               pipefd[2];
  pid_t             child;
  char              buffer;
  char             *message = "ciao a tutti voi amici "
                              "vicini e lontani\n";
  int               i;
  size_t            size;
  ssize_t           written;
  //
```

```
    //
    //
    if (pipe (pipefd) == -1)
      {
        perror ("pipe");
        exit (EXIT_FAILURE);
      }
    //
    //
    //
    child = fork ();
    if (child == -1)
      {
        perror ("fork");
        exit (EXIT_FAILURE);
      }
    //
    //
    //
    if (child == 0)
      {
        //
        // Questo è il figlio e deve leggere dal condotto;
        // pertanto, chiude il descrittore di scrittura nel
        // condotto.
        //
        close (pipefd[1]);
        //
        // Legge un byte alla volta, finché c'è qualcosa da
        // poter leggere.
        //
        while (read (pipefd[0], &buffer, 1) > 0)
          {
            //
```

```
            // Emette il byte letto attraverso lo standard
            // output.
            //
            write (STDOUT_FILENO, &buffer, 1);
          }
        //
        // Chiude il condotto e conclude il suo funzionamento.
        //
        close (pipefd[0]);
        //
        exit (EXIT_SUCCESS);
      }
    else
      {
        //
        // Questo è il genitore, il quale chiude il
        // descrittore di lettura nel condotto.
        //
        close (pipefd[0]);
        //
        // Inizia un ciclo senza fine.
        //
        while (1)
          {
            //
            // Scrive il messaggio contenuto nella stringa
            // 'message' attraverso il condotto.
            //
            for (i = 0, written = 0, size = strlen (message);
                 i < strlen (message);
                 i += written, size -= written)
              {
                written = write (pipefd[1], &message[i],
                                 size);
```

```
                if (written < 0)
                  {
                    //
                    // Essendosi verificato un errore, chiude
                    // il descrittore del condotto e si mette
                    // in attesa della morte del proprio
                    // processo figlio. Al termine conclude il
                    // proprio funzionamento.
                    //
                    perror ("pipe");
                    close (pipefd[1]);
                    wait (NULL);   // Wait for child.
                    exit (EXIT_FAILURE);
                  }
              }
          }
        //
        // Dal momento che il codice precedente è racchiuso in
        // un ciclo infinito, ciò che segue non può essere mai
        // eseguito.
        // Comunque, nel caso si volesse gestire il ciclo
        // precedente, a questo punto verrebbe chiuso il
        // condotto da parte del processo genitore, attendendo
        // la morte del proprio processo figlio, prima di
        // concludere regolarmente il proprio funzionamento.
        //
        close (pipefd[1]);
        wait (NULL);
        exit (EXIT_SUCCESS);
      }
}
```

Il programma dell’esempio, sdoppiandosi in due processi, da un lato (quello del genitore) scrive nel condotto la stringa ***message***

una quantità di volte indefinita, mentre dall'altro legge dal condotto il messaggio riproducendolo attraverso lo standard output sullo schermo, fino a quando uno dei due processi viene interrotto.

68.9.5 Esempio di condotto attraverso un file FIFO

«

Nella sezione precedente appare un esempio completo di programma che crea e utilizza un condotto. Nel listato successivo, disponibile eventualmente presso *allegati/c/esempio-posix-fifo.c* , si vede un altro esempio, pressocché equivalente, in cui i due processi prodotti comunicano attraverso un condotto derivante da un file FIFO. Il file FIFO viene creato preventivamente dal processo genitore, prima di sdoppiarsi, poi i due processi aprono il file e riproducono lo stesso comportamento già descritto nella sezione precedente.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
//----------------------------------------------------------
int
main (void)
{
  int              fd;
  pid_t            child;
  char             buffer;
  char            *message = "ciao a tutti voi amici "
                             "vicini e lontani\n";
  int              i;
  size_t           size;
```

```
    ssize_t          written;
    int              status;
    //
    // Prima di creare il file FIFO, cancella quello che
    // potrebbe esserci già.
    //
    unlink ("/tmp/fifo");
    //
    // Usa 'mknod()', ma potrebbe usare 'mkfifo()' per creare
    // il file FIFO.
    //
    status = mknod ("/tmp/fifo",
                    S_IFIFO | S_IRUSR | S_IWUSR, 0);
    if (status != 0)
      {
        perror ("mknod fifo");
        exit (EXIT_FAILURE);
      }
    //
    // Il processo si sdoppia.
    //
    child = fork ();
    if (child == -1)
      {
        perror ("fork");
        exit (EXIT_FAILURE);
      }
    //
    //
    //
    if (child == 0)
      {
        //
        // Questo è il processo figlio che apre il file FIFO
```

```
        // in lettura.
        //
        fd = open ("/tmp/fifo", O_RDONLY);
        if (fd < 0)
          {
            perror ("fifo read open");
            exit (EXIT_FAILURE);
          }
        //
        // Legge un byte alla volta riproducendolo attraverso
        // lo standard output.
        //
        while (read (fd, &buffer, 1) > 0)
          {
            write (STDOUT_FILENO, &buffer, 1);
          }
        //
        // Chiude il file FIFO e si conclude il funzionamento
        // del processo figlio.
        //
        close (fd);
        //
        exit (EXIT_SUCCESS);
      }
    else
      {
        //
        // Questo è il processo genitore che apre il file
        // FIFO in scrittura.
        //
        fd = open ("/tmp/fifo", O_WRONLY);
        if (fd < 0)
          {
            perror ("fifo write open");
```

```
            exit (EXIT_FAILURE);
          }
        //
        // Inizia un ciclo infinito.
        //
        while (1)
          {
            //
            // Scrive il messaggio contenuto nella stringa
            // `message' attraverso il condotto.
            //
            for (i = 0, written = 0, size = strlen (message);
                 i < strlen (message);
                 i += written, size -= written)
              {
                written = write (fd, &message[i], size);
                if (written < 0)
                  {
                    //
                    // Essendosi verificato un errore, chiude
                    // il file FIFO e si mette in attesa della
                    // morte del proprio processo figlio.
                    // Al termine conclude il proprio
                    // funzionamento.
                    //
                    perror ("pipe");
                    close (fd);
                    wait (NULL);
                    exit (EXIT_FAILURE);
                  }
              }
          }
        //
        // Dal momento che il codice precedente è racchiuso
```

```
        // in un ciclo infinito, ciò che segue non può essere
        // mai eseguito.
        // Comunque, nel caso si volesse gestire il ciclo
        // precedente, a questo punto verrebbe chiuso il file
        // FIFO da parte del processo genitore, attendendo la
        // morte del proprio processo figlio, prima di
        // concludere regolarmente il proprio funzionamento.
        //
        close (fd);
        wait (NULL);
        exit (EXIT_SUCCESS);
    }
}
```

68.10 Lettura delle directory

«

Le directory si creano con la funzione ***mkdir()***, in modo da garantire che le voci obbligatorie '`.`' e '`..`' siano presenti. Successivamente, l'aggiornamento delle directory avviene in modo trasparente da parte del sistema operativo, in base alle operazioni di creazione ed eliminazione dei file. Rimane il problema della lettura delle directory, dalla quale ottenere nomi e riferimenti a inode, per poter conoscere il loro «contenuto».

Di norma è possibile leggere le directory come se fossero dei file puri e semplici, ma così facendo occorre interpretarne il contenuto secondo le regole di quel file system particolare. In generale ciò è sconsigliabile, pertanto vengono in aiuto alcune funzioni descritte nel file di intestazione '`dirent.h`' (si veda anche la sezione 70.6 sul file di intestazione '`dirent.h`').

68.10.1 Tipi derivati

La gestione delle directory, secondo il file di intestazione '`dirent.h`', prevede due tipi derivati: '**`DIR`**' e '**`struct dirent`**'. Il tipo '**`DIR`**' serve a rappresentare una variabile strutturata con tutte le informazioni relative a un flusso di file riferito a una directory. In altri termini, è l'equivalente del tipo '**`FILE`**', ma utile solo per l'accesso alle directory. Il tipo '**`struct dirent`**' serve a poter rappresentare i due componenti indispensabili di ogni voce di directory dei sistemi Unix: numero di inode e nome. Pertanto, il tipo '**`struct dirent`**' contiene almeno i membri '**`d_ino`**' e '**`d_name`**', per contenere rispettivamente il numero di inode e il nome relativo:

```
struct dirent {
    ino_t   d_ino;
    char    d_name[];
    …
};
```

68.10.2 Procedura per accedere a una directory

Per accedere a una directory, occorre prima aprirla, con la funzione ***opendir()***, la quale restituisce un puntatore a una variabile di tipo '**`DIR`**', dove tale puntatore rappresenta così la directory in forma di flusso.

```
DIR *opendir (const char *path);
```

La lettura di una directory avviene a blocchi di una voce per volta, utilizzando la funzione ***readdir()***: ogni lettura produce la voce

successiva della directory, in forma di variabile strutturata di tipo '**`struct dirent`**', della quale si ottiene il puntatore.

```
struct dirent *readdir (DIR *dp);
```

Evidentemente, da come è strutturato il prototipo della funzione, si intuisce che la variabile strutturata a cui punta ciò che restituisce la funzione stessa, deve trovarsi in una zona di memoria statica, la quale viene riutilizzata ogni volta che si chiama la funzione ***readdir()***.

Dato che le letture si susseguono in modo sequenziale, quando si vuole che la prossima lettura ricominci dalla prima voce, si utilizza la funzione ***rewinddir()***:

```
void rewinddir (DIR *dp);
```

Al termine, come per i file normali, il flusso aperto di directory va chiuso con la funzione ***closedir()***, dove il valore restituito rappresenta il successo o meno dell'operazione:

```
int closedir (DIR *dp);
```

68.10.3 Attributo «FD_CLOEXEC»

«

Un sistema Unix dispone di un metodo di accesso ai file basato sui descrittori, sopra il quale si inserisce la gestione dei flussi di file. Per quanto riguarda le directory, lo standard non specifica se i flussi relativi debbano avvalersi dei descrittori o meno. Tuttavia, se si usano i descrittori, si presenta una situazione particolare: se si esegue un

altro processo, in sostituzione di quello in corso (con una delle funzioni ***exec...()***), i descrittori aperti vengono ereditati tali e quali dal nuovo processo. Nel caso delle directory, ciò va evitato.

Nei sistemi in cui il tipo '**`DIR`**' viene gestito tramite riferimenti a descrittori, l'apertura di una directory comporta l'attivazione dell'indicatore rappresentato dalla macro-variabile ***FD_CLOEXEC*** (file di intestazione '`fcntl.h`'), con il quale si assicura che il descrittore venga chiuso nel caso di utilizzo di una funzione ***exec...()***.

68.10.4 Esempio di utilizzo delle funzioni di accesso alle directory

«

Il listato successivo mostra un esempio molto semplice di programma che legge la directory corrente, mostrando l'elenco dei nomi che contiene, senza indicare però altre informazioni. Eventualmente si può ottenere il file dell'esempio dall'indirizzo *allegati/c/esempio-posix-dirent.c*

```
#include <errno.h>
#include <stdio.h>
#include <dirent.h>
//----------------------------------------------------------
int
main (int argc, char *argv[], char *envp[])
{
  DIR             *dp;
  struct dirent   *dir;
  //
  dp = opendir (".");
  if (dp == NULL)
    {
      perror (NULL);
      return (1);
    }
  //
  while ((dir = readdir (dp)) != NULL)
    {
      printf ("%s\n", dir->d_name);
    }
  //
  closedir (dp);
  //
  return (0);
}
```

Eventualmente si veda una realizzazione molto semplice del programma '**ls**' nei sorgenti di os32 (listato 96.1.23).

68.11 Riferimenti

- Maurice J. Bach, *The design of the UNIX operating system*, Prentice Hall, 1990, ISBN 0132017997

- The Open Group, *The UNIX System*, http://www.unix.org/
- Free Software Foundation, *The GNU C Library*, http://www.gnu.org/software/libc/manual/
- The Open Group, *The Single UNIX® Specification, Version 2*, *Regular Expressions*, http://pubs.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap09.html
- Daniel Robbins, *POSIX threads explained*, http://www.ibm.com/developerworks/library/l-posix1/ , http://www.ibm.com/developerworks/library/l-posix2/ , http://www.ibm.com/developerworks/library/l-posix3/
- pagina di manuale *pthreads(7)* di un sistema GNU/Linux
- Mark Hayes, *POSIX threads tutorial*, http://math.arizona.edu/~swig/documentation/pthreads/
- The Open Group, *The Single UNIX® Specification, Version 2*, *pthread.h*, http://pubs.opengroup.org/onlinepubs/000095399/basedefs/pthread.h.html
- The Open Group, *The Single UNIX® Specification, Version 2*, *dirent.h*, http://pubs.opengroup.org/onlinepubs/000095399/basedefs/dirent.h.html
- Pagine di manuale: *mknod(2)*, *mkdir(2)*, *mkfifo(3)*, *unlink(2)*, *pipe(2)*, *mkfifo(3)*, *opendir(3)*, *readdir(3)*, *rewinddir(3)*, *closedir(3)*

[1] Per la libreria POSIX, la gestione dei flussi del linguaggio C è costruita avvalendosi del sistema dei descrittori, con l'aggiunta però di una memoria tampone.

[2] Nel programma di esempio si può fare sicuramente di meglio, incrementando direttamente la variabile globale, senza tanti travasi come invece viene fatto. Ma lo scopo di questi esempi è simulare una situazione più complessa, senza complicazioni che esulano dal problema specifico che si vuole descrivere.