## os16: directory principale

«

### bochs

«

Si veda la sezione u0.2.

```
10001  #!/bin/sh
10002
10003  bochs -q "boot:floppy" \
10004           "floppya: 1_44=floppy.a, status=inserted" \
10005           "floppyb: 1_44=floppy.b, status=inserted" \
10006           "keyboard_mapping: enabled=1, \
10007            map=/usr/share/bochs/keymaps/x11-pc-it.map" \
10008           "keyboard_type: xt" \
10009           "vga: none" \
10010           "romimage: file=\"/usr/share/bochs/BIOS-bochs-legacy\"" \
10011           "megs:1"
```

### qemu

«

Si veda la sezione u0.2.

```
20001  #!/bin/sh
20002
20003  qemu -fda floppy.a \
20004       -fdb floppy.b \
20005       -boot order=a
20006
```

### makeit

«

Si veda la sezione u0.2.

```
30001  #!/bin/sh
30002  #
30003  # makeit...
30004  #
30005  OPTION="$1"
30006  OS16PATH=""
30007  #
30008  edition () {
30009      local EDITION="kernel/main/build.h"
30010      echo -n                                > $EDITION
30011      echo -n "#define BUILD_DATE \""      >> $EDITION
30012      echo -n `date "+%Y.%m.%d %H:%M:%S"` >> $EDITION
30013      echo  "\""                           >> $EDITION
30014  }
30015  #
30016  #
30017  #
30018  makefile () {
30019      #
30020      local MAKEFILE="Makefile"
30021      local TAB="	"
30022      #
30023      local SOURCE_C=""
30024      local C=""
30025      local SOURCE_S=""
```

```
30026     local S=""
30027     #
30028     local c
30029     local s
30030     #
30031     # Trova i file in C.
30032     #
30033     for c in *.c
30034     do
30035         if [ -f $c ]
30036         then
30037         C=‘basename $c .c‘
30038             SOURCE_C="$SOURCE_C $C"
30039         fi
30040     done
30041     #
30042     # Trova i file in ASM.
30043     #
30044     for s in *.s
30045     do
30046         if [ -f $s ]
30047         then
30048         S=‘basename $s .s‘
30049             SOURCE_S="$SOURCE_S $S"
30050         fi
30051     done
30052     #
30053     # Prepara il file make.
30054     # GCC viene usato per potenziare il controllo degli errori.
30055     #
30056     echo -n                                                  > $MAKEFILE
30057     echo "# This file was made automatically"               >> $MAKEFILE
30058     echo "# by the script \‘makeit\’, based on the"         >> $MAKEFILE
30059     echo "# directory content."                             >> $MAKEFILE
30060     echo "# Please use \‘makeit\’ to compile and"          >> $MAKEFILE
30061     echo "# \‘makeit clean\’ to clean directories."         >> $MAKEFILE
30062     echo "#"                                              >> $MAKEFILE
30063     echo "c = $SOURCE_C"                                  >> $MAKEFILE
30064     echo "#"                                              >> $MAKEFILE
30065     echo "s = $SOURCE_S"                                  >> $MAKEFILE
30066     echo "#"                                              >> $MAKEFILE
30067     echo "all: \$(s) \$(c)"                               >> $MAKEFILE
30068     echo "#"                                              >> $MAKEFILE
30069     echo "clean:"                                         >> $MAKEFILE
30070     echo "${TAB}@rm \$(c) \$(s) *.o *.assembler 2> /dev/null ; true" \
30071                                                           >> $MAKEFILE
30072     echo "${TAB}@rm *.symbols 2> /dev/null     ; true"    >> $MAKEFILE
30073     echo "${TAB}@pwd"                                     >> $MAKEFILE
30074     echo "#"                                              >> $MAKEFILE
30075     echo "\$(c):"                                         >> $MAKEFILE
30076     echo "${TAB}@echo \$@.c"                              >> $MAKEFILE
30077     echo "${TAB}@gcc -Wall -c -o \$@.o " \
30078                         "-I " \
30079                         "-I. " \
30080                         "-I$OS16PATH/lib " \
30081                         "-I$OS16PATH/ " \
30082                         "\$@.c"                           >> $MAKEFILE
30083     echo "${TAB}@rm \$@.o"                                >> $MAKEFILE
30084     echo "${TAB}@bcc -ansi -0 -Mc -S -o \$@.assembler " \
30085                         "-I " \
30086                         "-I. " \
30087                         "-I$OS16PATH/lib " \
30088                         "-I$OS16PATH/ " \
30089                         "\$@.c"                           >> $MAKEFILE
30090     echo "${TAB}@bcc -ansi -0 -Mc -c -o \$@.o " \
30091                         "-I " \
30092                         "-I. " \
30093                         "-I$OS16PATH/lib " \
30094                         "-I$OS16PATH/ " \
30095                         "\$@.c"                           >> $MAKEFILE
30096     echo "#"                                              >> $MAKEFILE
30097     echo "\$(s):"                                         >> $MAKEFILE
30098     echo "${TAB}@echo \$@.s"                              >> $MAKEFILE
30099     echo "${TAB}@as86 -u -0 -o \$@.o -s \$@.symbols \$@.s"  >> $MAKEFILE
30100     #
30101 }
30102 #
30103 #
30104 #
30105 main () {
30106     #
30107     local CURDIR=‘pwd‘
30108     local OBJECTS
30109     local OBJLIB
30110     local EXEC
30111     local BASENAME
30112     local PROGNAME
30113     local d
30114     local c
30115     local s
30116     local o
30117     #
30118     edition
30119     #
30120     # Copia dello scheletro
30121     #
30122     if [ "$OPTION" = "clean" ]
30123     then
30124         #
30125         # La copia non va fatta.
30126         #
```



```
30127         true
30128     else
30129         cp -dpRv skel/etc   /mnt/os16.a/
30130         cp -dpRv skel/dev   /mnt/os16.a/
30131         mkdir               /mnt/os16.a/mnt/
30132         mkdir               /mnt/os16.a/tmp/
30133         chmod 0777          /mnt/os16.a/tmp/
30134         mkdir               /mnt/os16.a/usr/
30135         cp -dpRv skel/root  /mnt/os16.a/
30136         cp -dpRv skel/home  /mnt/os16.a/
30137         cp -dpRv skel/usr/* /mnt/os16.b/
30138     fi
30139     #
30140     #
30141     #
30142     for d in ‘find kernel‘  \
30143              ‘find lib‘     \
30144              ‘find applic‘  \
30145              ‘find ported‘
30146     do
30147         if [ -d "$d" ]
30148         then
30149             #
30150             # Sono presenti dei file C o ASM?
30151             #
30152             c=‘echo $d/*.c | sed "s/ .*//"‘
30153             s=‘echo $d/*.s | sed "s/ .*//"‘
30154             #
30155             if [ -f "$c" ] || [ -f "$s" ]
30156             then
30157                 #
30158                 # Sì
30159                 #
30160                 CURDIR=‘pwd‘
30161                 cd $d
30162                 #
30163                 # Ricrea il file make
30164                 #
30165                 makefile
30166                 #
30167                 # Pulisce quindi la directory
30168                 #
30169                 make clean
30170                 #
30171                 #
30172                 #
30173                 if [ "$OPTION" = "clean" ]
30174                 then
30175                     #
30176                     # È stata richiesta la pulitura, ma questa
30177                     # è appena stata fatta!
30178                     #
30179                     true
30180                 else
30181                     #
30182                     # Qualunque altro argomento viene considerato
30183                     # un ‘make‘.
30184                     #
30185                     if ! make
30186                     then
30187                         #
30188                         # La compilazione è fallita.
30189                         #
30190                         cd "$CURDIR"
30191                         exit
30192                     fi
30193                 fi
30194                 cd "$CURDIR"
30195             fi
30196         fi
30197     done
30198     #
30199     cd "$CURDIR"
30200     #
30201     # Link
30202     #
30203     if [ "$OPTION" = "clean" ]
30204     then
30205         #
30206         # Il collegamento non va fatto.
30207         #
30208         true
30209     else
30210         #
30211         # Collegamento dei file del kernel.
30212         #
30213         OBJECTS=""
30214         #
30215         for o in ‘find kernel   -name \*.o -print‘ \
30216                  ‘find lib      -name \*.o -print‘
30217         do
30218             if    [ "$o" = "./kernel/main/crt0.o" ] \
30219                || [ "$o" = "./kernel/main/main.o" ] \
30220                || [ ! -e "$o" ]
30221             then
30222                 true
30223             else
30224                 OBJECTS="$OBJECTS $o"
30225             fi
30226         done
30227         #
```

```
30228         echo "Link"
30229         #
30230         ld86 -i -d -s -m -o kimage    \
30231                 kernel/main/crt0.o    \
30232                 kernel/main/main.o    \
30233                 $OBJECTS
30234         #
30235         # Copia il kernel nel dischetto.
30236         #
30237         if mount | grep /mnt/os16.a > /dev/null
30238         then
30239             cp -f kimage /mnt/os16.a/boot
30240         else
30241             echo "[$0] Cannot copy the kernel image "
30242             echo "[$0]   inside the floppy disk image!"
30243         fi
30244         sync
30245         #
30246         # Collegamento delle applicazioni di os16.
30247         #
30248         OBJLIB=""
30249         #
30250         for o in `find lib      -name \*.o -print`
30251         do
30252             OBJLIB="$OBJLIB $o"
30253         done
30254         #
30255         # Scansione delle applicazioni interne.
30256         #
30257         for o in `find applic     -name \*.o -print`
30258         do
30259             if    [ "$o" = "applic/crt0.o" ] \
30260                || [ ! -e "$o" ] \
30261                || echo "$o" | grep ".crt0.o$" > /dev/null
30262             then
30263                 #
30264                 # Il file non esiste oppure si tratta di `...crt0.s'.
30265                 #
30266                 true
30267             else
30268                 #
30269                 # File oggetto differente da `...crt0.s'.
30270                 #
30271                 EXEC=`echo "$o" | sed "s/\.o$//"`
30272                 BASENAME=`basename $o .o`
30273                 if [ -e "applic/$BASENAME.crt0.o" ]
30274                 then
30275                     #
30276                     # Qui c'è un file `...crt0.o' specifico.
30277                     #
30278                     ld86 -i -d -s -o $EXEC \
30279                          applic/$BASENAME.crt0.o $o $OBJLIB
30280                 else
30281                     #
30282                     # Qui si usa il file `crt0.o' generale.
30283                     #
30284                     ld86 -i -d -s -o $EXEC applic/crt0.o $o $OBJLIB
30285                 fi
30286                 #
30287                 if [ -x "applic/$BASENAME" ]
30288                 then
30289                     if mount | grep /mnt/os16.a > /dev/null
30290                     then
30291                         mkdir /mnt/os16.a/bin/ 2> /dev/null
30292                         cp -f "$EXEC" /mnt/os16.a/bin
30293                     else
30294                         echo "[$0] Cannot copy the application "
30295                         echo "[$0]   $BASENAME inside the floppy "
30296                         echo "[$0]   disk image!"
30297                         break
30298                     fi
30299                 fi
30300             fi
30301         done
30302         sync
30303         #
30304         # Collegamento delle applicazioni più semplici,
30305         # provenienti da altri sistemi operativi.
30306         #
30307         for o in `find ported/mix -name \*.o -print`
30308         do
30309             if    [ "$o" = "ported/mix/crt0.o" ] \
30310                || [ ! -e "$o" ] \
30311                || echo "$o" | grep ".crt0.o$" > /dev/null
30312             then
30313                 #
30314                 # Il file non esiste oppure si tratta di `...crt0.s'.
30315                 #
30316                 true
30317             else
30318                 #
30319                 # File oggetto differente da `...crt0.s'.
30320                 #
30321                 EXEC=`echo "$o" | sed "s/\.o$//"`
30322                 BASENAME=`basename $o .o`
30323                 if [ -e "ported/mix/$BASENAME.crt0.o" ]
30324                 then
30325                     #
30326                     # Qui c'è un file `...crt0.o' specifico.
30327                     #
30328                     ld86 -i -d -s -o $EXEC \
```



```
30329                          applic/$BASENAME.crt0.o $o $OBJLIB
30330                 else
30331                     #
30332                     # Qui si usa il file `crt0.o' generale.
30333                     #
30334                     ld86 -i -d -s -o $EXEC applic/crt0.o $o $OBJLIB
30335                 fi
30336                 #
30337                 if [ -x "$EXEC" ]
30338                 then
30339                     if mount | grep /mnt/os16.a > /dev/null
30340                     then
30341                         mkdir /mnt/os16.b/bin/ 2> /dev/null
30342                         cp -f "$EXEC" /mnt/os16.b/bin
30343                     else
30344                         echo "[$0] Cannot copy the application "
30345                         echo "[$0]   $EXEC inside the floppy "
30346                         echo "[$0]   disk image!"
30347                         break
30348                     fi
30349                 fi
30350             fi
30351         done
30352         sync
30353         #
30354         # Altre applicazioni più importanti.
30355         #
30356         for d in ported/*
30357         do
30358             if [ -d "$d" ]
30359             then
30360                 #
30361                 #
30362                 #
30363                 OBJECTS=""
30364                 BASENAME=`basename $d`
30365                 EXEC="$d/$BASENAME"
30366                 #
30367                 #
30368                 #
30369                 if [ "$BASENAME" = "mix" ]
30370                 then
30371                     #
30372                     # già fatto.
30373                     #
30374                     continue
30375                 fi
30376                 #
30377                 #
30378                 #
30379                 for o in $d/*.o
30380                 do
30381                     if    [ "$o" = "$d/crt0.o" ] \
30382                        || [ ! -e "$o" ]
30383                     then
30384                         true
30385                     else
30386                         OBJECTS="$OBJECTS $o"
30387                     fi
30388                 done
30389                 #
30390                 ld86 -i -d -s -o $EXEC $d/crt0.o $OBJECTS $OBJLIB
30391                 #
30392                 if [ -x "$d/$BASENAME" ]
30393                 then
30394                     if mount | grep /mnt/os16.b > /dev/null
30395                     then
30396                         mkdir /mnt/os16.b/bin/ 2> /dev/null
30397                         cp -f "$EXEC" /mnt/os16.b/bin
30398                     else
30399                         echo "[$0] Cannot copy the application "
30400                         echo "[$0]   $BASENAME inside the floppy "
30401                         echo "[$0]   disk image!"
30402                         break
30403                     fi
30404                 fi
30405             fi
30406         done
30407         sync
30408
30409     fi
30410 }
30411 #
30412 # Start.
30413 #
30414 if [ -d kernel ]   && \
30415    [ -d applic ]   && \
30416    [ -d lib  ]
30417 then
30418     OS16PATH=`pwd`
30419     main
30420 else
30421     echo "[$0] Running from a wrong directory!"
30422 fi
```

## os16: «kernel/devices.h»

«

Si veda la sezione u0.1.

```
40001  #ifndef _KERNEL_DEVICES_H
40002  #define _KERNEL_DEVICES_H 1
40003
40004  #include <sys/os16.h>
40005  #include <sys/types.h>
40006  //-------------------------------------------------------------------------
40007  #define DEV_READ                0
40008  #define DEV_WRITE               1
40009  ssize_t dev_io    (pid_t pid, dev_t device, int rw, off_t offset,
40010                     void *buffer, size_t size, int *eof);
40011  //-------------------------------------------------------------------------
40012  // The following functions are used only by 'dev_io()'.
40013  //-------------------------------------------------------------------------
40014  ssize_t dev_mem   (pid_t pid, dev_t device, int rw, off_t offset,
40015                     void *buffer, size_t size, int *eof);
40016  ssize_t dev_tty   (pid_t pid, dev_t device, int rw, off_t offset,
40017                     void *buffer, size_t size, int *eof);
40018  ssize_t dev_dsk   (pid_t pid, dev_t device, int rw, off_t offset,
40019                     void *buffer, size_t size, int *eof);
40020  ssize_t dev_kmem  (pid_t pid, dev_t device, int rw, off_t offset,
40021                     void *buffer, size_t size, int *eof);
40022  //-------------------------------------------------------------------------
40023
40024  #endif
```

## kernel/devices/dev_dsk.c

«

Si veda la sezione i159.1.2.

```
50001  #include <sys/os16.h>
50002  #include <kernel/devices.h>
50003  #include <sys/types.h>
50004  #include <errno.h>
50005  #include <kernel/memory.h>
50006  #include <kernel/ibm_i86.h>
50007  #include <kernel/proc.h>
50008  #include <string.h>
50009  #include <signal.h>
50010  #include <kernel/k_libc.h>
50011  #include <ctype.h>
50012  #include <kernel/tty.h>
50013  //-------------------------------------------------------------------------
50014  ssize_t
50015  dev_dsk (pid_t pid, dev_t device, int rw, off_t offset, void *buffer,
50016           size_t size, int *eof)
50017  {
50018      ssize_t n;
50019      int     dev_minor = minor (device);
50020
50021      if (rw == DEV_READ)
50022        {
50023          n = dsk_read_bytes (dev_minor, offset, buffer, size);
50024        }
50025      else
50026        {
50027          n = dsk_write_bytes (dev_minor, offset, buffer, size);
50028        }
50029      return (n);
50030  }
```

## kernel/devices/dev_io.c

«

Si veda la sezione i159.1.1.

```
60001  #include <sys/os16.h>
60002  #include <kernel/devices.h>
60003  #include <sys/types.h>
60004  #include <errno.h>
60005  #include <kernel/memory.h>
60006  #include <kernel/ibm_i86.h>
60007  #include <kernel/proc.h>
60008  #include <string.h>
60009  #include <signal.h>
60010  #include <kernel/k_libc.h>
60011  #include <ctype.h>
60012  #include <kernel/tty.h>
60013  //-------------------------------------------------------------------------
60014  ssize_t
60015  dev_io (pid_t pid, dev_t device, int rw, off_t offset,
60016          void *buffer, size_t size, int *eof)
60017  {
60018      int dev_major = major (device);
60019      if (rw != DEV_READ && rw != DEV_WRITE)
60020        {
60021          errset (EIO);
60022          return (-1);
60023        }
60024      switch (dev_major)
60025        {
60026          case DEV_MEM_MAJOR:
60027              return (dev_mem (pid, device, rw, offset, buffer, size,
60028                               eof));
60029          case DEV_TTY_MAJOR:
60030              return (dev_tty (pid, device, rw, offset, buffer, size,
60031                               eof));
60032          case DEV_CONSOLE_MAJOR:
60033              return (dev_tty (pid, device, rw, offset, buffer, size,
60034                               eof));
60035          case DEV_DSK_MAJOR:
60036              return (dev_dsk (pid, device, rw, offset, buffer, size,
60037                               eof));
60038          case DEV_KMEM_MAJOR:
60039              return (dev_kmem (pid, device, rw, offset, buffer, size,
60040                                eof));
60041          default:
60042              errset (ENODEV);
60043              return (-1);
60044        }
60045  }
```



## kernel/devices/dev_kmem.c

«

Si veda la sezione i159.1.3.

```
70001  #include <sys/os16.h>
70002  #include <kernel/devices.h>
70003  #include <sys/types.h>
70004  #include <errno.h>
70005  #include <kernel/memory.h>
70006  #include <kernel/ibm_i86.h>
70007  #include <kernel/proc.h>
70008  #include <string.h>
70009  #include <signal.h>
70010  #include <kernel/k_libc.h>
70011  #include <ctype.h>
70012  #include <kernel/tty.h>
70013  //-------------------------------------------------------------------------
70014  ssize_t
70015  dev_kmem (pid_t pid, dev_t device, int rw, off_t offset, void *buffer,
70016            size_t size, int *eof)
70017  {
70018    size_t    size_real;
70019    inode_t  *inode;
70020    sb_t     *sb;
70021    file_t   *file;
70022    void     *start;
70023    //
70024    // Only read is allowed.
70025    //
70026    if (rw != DEV_READ)
70027      {
70028        errset (EIO);                           // I/O error.
70029        return ((ssize_t) -1);
70030      }
70031    //
70032    // Only positive offset is allowed.
70033    //
70034    if (offset < 0)
70035      {
70036        errset (EIO);                           // I/O error.
70037        return ((ssize_t) -1);
70038      }
70039    //
70040    // Read is selected (and is the only access allowed).
70041    //
70042    switch (device)
70043      {
70044        case DEV_KMEM_PS:
70045            //
70046            // Verify if the selected slot can be read.
70047            //
70048            if (offset >= PROCESS_MAX)
70049              {
70050                errset (EIO);                   // I/O error.
70051                return ((ssize_t) -1);
70052              }
70053            //
70054            // Correct the size to be read.
70055            //
70056            if (sizeof (proc_t) < size)
70057              {
70058                size = sizeof (proc_t);
70059              }
70060            //  //
70061            //  // Correct the size to be read.
70062            //  //
70063            //  size_real = ((sizeof (proc_t)) * (PROCESS_MAX - offset));
70064            //  if (size_real < size)
70065            //    {
70066            //      size = size_real;
70067            //    }
70068            //
70069            // Get the pointer to the selected slot.
70070            //
70071            start = proc_reference ((pid_t) offset);
70072            break;
70073        case DEV_KMEM_MMP:
70074            //
70075            // Correct the size to be read.
70076            //
70077            size_real = (MEM_MAX_BLOCKS/8);
70078            if (size_real < size)
70079              {
70080                size = size_real;
70081              }
70082            //
70083            // Get the pointer to the map.
```

```
70084           //
70085           start = mb_reference ();
70086           break;
70087       case DEV_KMEM_SB:
70088           //
70089           // Get a reference to the super block table.
70090           //
70091           sb = sb_reference (0);
70092           //
70093           // Correct the size to be read.
70094           //
70095           if (sizeof (sb_t) < size)
70096             {
70097               size = sizeof (sb_t);
70098             }
70099           //
70100           // Get the pointer to the selected super block slot.
70101           //
70102           start = &sb[offset];
70103           break;
70104       case DEV_KMEM_INODE:
70105           //
70106           // Get a reference to the inode table.
70107           //
70108           inode = inode_reference (0, 0);
70109           //
70110           // Correct the size to be read.
70111           //
70112           if (sizeof (inode_t) < size)
70113             {
70114               size = sizeof (inode_t);
70115             }
70116           //
70117           // Get the pointer to the selected inode slot.
70118           //
70119           start = &inode[offset];
70120           break;
70121       case DEV_KMEM_FILE:
70122           //
70123           // Get a reference to the file table.
70124           //
70125           file = file_reference (0);
70126           //
70127           // Correct the size to be read.
70128           //
70129           if (sizeof (file_t) < size)
70130             {
70131               size = sizeof (file_t);
70132             }
70133           //
70134           // Get the pointer to the selected inode slot.
70135           //
70136           start = &file[offset];
70137           break;
70138       default:
70139           errset (ENODEV);            // No such device.
70140           return ((ssize_t) -1);
70141     }
70142   //
70143   // At this point, data is ready to be copied to the buffer.
70144   //
70145   memcpy (buffer, start, size);
70146   //
70147   // Return size read.
70148   //
70149   return (size);
70150 }
```

## kernel/devices/dev_mem.c

«

Si veda la sezione i159.1.4.

```
80001 #include <sys/os16.h>
80002 #include <kernel/devices.h>
80003 #include <sys/types.h>
80004 #include <errno.h>
80005 #include <kernel/memory.h>
80006 #include <kernel/ibm_i86.h>
80007 #include <kernel/proc.h>
80008 #include <string.h>
80009 #include <signal.h>
80010 #include <kernel/k_libc.h>
80011 #include <ctype.h>
80012 #include <kernel/tty.h>
80013 //------------------------------------------------------------------------
80014 ssize_t
80015 dev_mem (pid_t pid, dev_t device, int rw, off_t offset, void *buffer,
80016          size_t size, int *eof)
80017 {
80018     uint8_t  *buffer08 = (uint8_t *) buffer;
80019     uint16_t *buffer16 = (uint16_t *) buffer;
80020     ssize_t n;
80021
80022     if      (device == DEV_MEM)                         // DEV_MEM
80023       {
80024         if (rw == DEV_READ)
80025           {
80026             n = mem_read ((addr_t) offset, buffer, size);
80027           }
80028         else
80029           {
```



```
80030             n = mem_write ((addr_t) offset, buffer, size);
80031           }
80032       }
80033     else if (device == DEV_NULL)                        // DEV_NULL
80034       {
80035         n = 0;
80036       }
80037     else if (device == DEV_ZERO)                        // DEV_ZERO
80038       {
80039         if (rw == DEV_READ)
80040           {
80041             for (n = 0; n < size; n++)
80042               {
80043                 buffer08[n] = 0;
80044               }
80045           }
80046         else
80047           {
80048             n = 0;
80049           }
80050       }
80051     else if (device == DEV_PORT)                        // DEV_PORT
80052       {
80053         if (rw == DEV_READ)
80054           {
80055             if (size == 1)
80056               {
80057                 buffer08[0] = in_8 (offset);
80058                 n = 1;
80059               }
80060             else if (size == 2)
80061               {
80062                 buffer16[0] = in_16 (offset);
80063                 n = 2;
80064               }
80065             else
80066               {
80067                 n = 0;
80068               }
80069           }
80070         else
80071           {
80072             if (size == 1)
80073               {
80074                 out_8 (offset, buffer08[0]);
80075               }
80076             else if (size == 2)
80077               {
80078                 out_16 (offset, buffer16[0]);
80079                 n = 2;
80080               }
80081             else
80082               {
80083                 n = 0;
80084               }
80085           }
80086       }
80087     else
80088       {
80089         errset (ENODEV);
80090         return (-1);
80091       }
80092     return (n);
80093 }
```

## kernel/devices/dev_tty.c

«

Si veda la sezione i159.1.5.

```
90001 #include <sys/os16.h>
90002 #include <kernel/devices.h>
90003 #include <sys/types.h>
90004 #include <errno.h>
90005 #include <kernel/memory.h>
90006 #include <kernel/ibm_i86.h>
90007 #include <kernel/proc.h>
90008 #include <string.h>
90009 #include <signal.h>
90010 #include <kernel/k_libc.h>
90011 #include <ctype.h>
90012 #include <kernel/tty.h>
90013 //------------------------------------------------------------------------
90014 ssize_t
90015 dev_tty (pid_t pid, dev_t device, int rw, off_t offset, void *buffer,
90016          size_t size, int *eof)
90017 {
90018     uint8_t *buffer08 = (uint8_t *) buffer;
90019     ssize_t  n;
90020     proc_t  *ps;
90021     //
90022     // Get process. Variable 'ps' will be 'NULL' if the process ID is
90023     // not valid.
90024     //
90025     ps = proc_reference (pid);
90026     //
90027     // Convert 'DEV_TTY' with the controlling terminal for the process.
90028     //
90029     if (device == DEV_TTY)
90030       {
90031         device = ps->device_tty;
90032         //
```

```
90033         // As a last resort, use the generic 'DEV_CONSOLE'.
90034         //
90035         if (device == 0 || device == DEV_TTY)
90036           {
90037             device = DEV_CONSOLE;
90038           }
90039       }
90040     //
90041     // Convert 'DEV_CONSOLE' to the currently active console.
90042     //
90043     if (device == DEV_CONSOLE)
90044       {
90045         device = tty_console ((dev_t) 0);
90046         //
90047         // As a last resort, use the first console: 'DEV_CONSOLE0'.
90048         //
90049         if (device == 0 || device == DEV_TTY)
90050           {
90051             device = DEV_CONSOLE0;
90052           }
90053       }
90054     //
90055     // Read or write.
90056     //
90057     if (rw == DEV_READ)
90058       {
90059         for (n = 0; n < size; n++)
90060           {
90061             buffer08[n] = tty_read (device);
90062             if (buffer08[n] == 0)
90063               {
90064                 //
90065                 // If the pid is not the kernel, should put the process
90066                 // to sleep, waiting for the key.
90067                 //
90068                 if (pid == 0 || ps == NULL)
90069                   {
90070                     //
90071                     // For the kernel there is no sleep and for an
90072                     // unidentified process, either.
90073                     //
90074                     break;
90075                   }
90076                 //
90077                 // Put the process to sleep.
90078                 //
90079                 ps->status         = PROC_SLEEPING;
90080                 ps->ret            = 0;
90081                 ps->wakeup_events  = WAKEUP_EVENT_TTY;
90082                 ps->wakeup_signal  = 0;
90083                 ps->wakeup_timer   = 0;
90084                 //
90085                 break;
90086               }
90087             //
90088             // Check for control characters.
90089             //
90090             if (buffer08[n] == 0x04)    // EOT
90091               {
90092                 //
90093                 // Return EOF.
90094                 //
90095                 *eof = 1;
90096                 break;
90097               }
90098             //
90099             // At this point, show the character on screen, even if it
90100             // is not nice. It is necessary to show something, because
90101             // the tty handling is very poor and the library for line
90102             // input, calculate cursor position based on the characters
90103             // received.
90104             //
90105             tty_write (device, (int) buffer08[n]);
90106           }
90107       }
90108     else
90109       {
90110         for (n = 0; n < size; n++)
90111           {
90112             tty_write (device, (int) buffer08[n]);
90113           }
90114       }
90115     return (n);
90116 }
```

## os16: «kernel/diag.h»

«

Si veda la sezione u0.2.

```
100001 #ifndef _KERNEL_DIAG_H
100002 #define _KERNEL_DIAG_H 1
100003
100004 #include <stdint.h>
100005 #include <kernel/fs.h>
100006 #include <sys/types.h>
100007 #include <kernel/proc.h>
100008
100009 //-------------------------------------------------------------------------
100010 uint8_t  reverse_8_bit  (uint8_t  source);
100011 uint16_t reverse_16_bit (uint16_t source);
100012 uint32_t reverse_32_bit (uint32_t source);
```



```
100013
100014 #define reverse_char(s)      ((char)     reverse_8_bit  ((uint8_t)  s))
100015 #define reverse_short(s)     ((short)    reverse_16_bit ((uint16_t) s))
100016 #define reverse_int(s)       ((int)      reverse_16_bit ((uint16_t) s))
100017 #define reverse_long(s)      ((long)     reverse_32_bit ((uint32_t) s))
100018 #define reverse_long_int(s)  ((long int) reverse_32_bit ((uint32_t) s))
100019 //-------------------------------------------------------------------------
100020 void print_hex_8  (void *data, size_t elements);
100021 void print_hex_16 (void *data, size_t elements);
100022 void print_hex_32 (void *data, size_t elements);
100023
100024 #define print_hex_char(d, e)      (print_hex_8  (d, e))
100025 #define print_hex_short(d, e)     (print_hex_16 (d, e))
100026 #define print_hex_int(d, e)       (print_hex_16 (d, e))
100027 #define print_hex_long(d, e)      (print_hex_32 (d, e))
100028 #define print_hex_long_int(d, e)  (print_hex_32 (d, e))
100029 //-------------------------------------------------------------------------
100030 void print_hex_8_reverse  (void *data, size_t elements);
100031 void print_hex_16_reverse (void *data, size_t elements);
100032 void print_hex_32_reverse (void *data, size_t elements);
100033
100034 #define print_hex_char_reverse(d, e)      (print_hex_8_reverse  (d, e))
100035 #define print_hex_short_reverse(d, e)     (print_hex_16_reverse (d, e))
100036 #define print_hex_int_reverse(d, e)       (print_hex_16_reverse (d, e))
100037 #define print_hex_long_reverse(d, e)      (print_hex_32_reverse (d, e))
100038 #define print_hex_long_int_reverse(d, e)  (print_hex_32_reverse (d, e))
100039 //-------------------------------------------------------------------------
100040 void print_segments (void);
100041 void print_kmem     (void);
100042 //-------------------------------------------------------------------------
100043 void print_mb_map     (void);
100044 void print_memory_map (void);
100045 //-------------------------------------------------------------------------
100046 void print_superblock       (sb_t *sb);
100047 void print_inode            (inode_t *inode);
100048 void print_inode_map        (sb_t *sb, uint16_t *bitmap);
100049 void print_zone_map         (sb_t *sb, uint16_t *bitmap);
100050 void print_inode_head       (void);
100051 void print_inode_list       (void);
100052 void print_inode_zones_head (void);
100053 void print_inode_zones      (inode_t *inode);
100054 void print_inode_zones_list (void);
100055 //-------------------------------------------------------------------------
100056 void print_proc_head (void);
100057 void print_proc_pid  (proc_t *ps, pid_t pid);
100058 void print_proc_list (void);
100059 //-------------------------------------------------------------------------
100060 void print_file_head (void);
100061 void print_file_num  (int num);
100062 void print_file_list (void);
100063 //-------------------------------------------------------------------------
100064 void print_fd_head (void);
100065 void print_fd      (fd_t *fd);
100066 void print_fd_list (pid_t pid);
100067 //-------------------------------------------------------------------------
100068 void print_time      (void);
100069 //-------------------------------------------------------------------------
100070
100071 #endif
```

## kernel/diag/print_fd.c

«

Si veda la sezione u0.2.

```
110001 #include <sys/os16.h>
110002 #include <kernel/diag.h>
110003 #include <kernel/k_libc.h>
110004 #include <fcntl.h>
110005 //-------------------------------------------------------------------------
110006 void
110007 print_fd (fd_t *fd)
110008 {
110009     k_printf ("%04x %6li %3i %c/%c %05o %5i %3i %5li %4i %04x %3i",
110010         (unsigned int)      fd->fl_flags,
110011         (unsigned long int) fd->file->offset,
110012         (unsigned int)      fd->file->references,
110013         (fd->file->oflags & O_RDONLY ? 'r' : ' '),
110014         (fd->file->oflags & O_WRONLY ? 'w' : ' '),
110015         (unsigned int)      fd->file->inode->mode,
110016         (unsigned int)      fd->file->inode->uid,
110017         (unsigned int)      fd->file->inode->gid,
110018         (unsigned long int) fd->file->inode->size,
110019         (unsigned int)      fd->file->inode->links,
110020         (unsigned int)      fd->file->inode->sb->device,
110021         (unsigned int)      fd->file->inode->ino);
110022     k_printf ("\n");
110023 }
```

## kernel/diag/print_fd_head.c

«

Si veda la sezione u0.2.

```
120001 #include <sys/os16.h>
120002 #include <kernel/diag.h>
120003 #include <kernel/k_libc.h>
120004 //-------------------------------------------------------------------------
120005 void
120006 print_fd_head (void)
120007 {
120008
```

```
120009      k_printf ("n. stat offset ref flg  mode   uid gid  size lnks  ");
120010      k_printf ("dev ino\n");
120011  }
```

kernel/diag/print_fd_list.c

«

Si veda la sezione u0.2.

```
130001  #include <sys/os16.h>
130002  #include <kernel/diag.h>
130003  #include <kernel/k_libc.h>
130004  //-----------------------------------------------------------------------
130005  void
130006  print_fd_list (pid_t pid)
130007  {
130008      int      fdn = 0;
130009      fd_t *fd;
130010      fd = fd_reference (pid, &fdn);
130011      print_fd_head ();
130012      for (fdn = 0; fdn < OPEN_MAX; fdn++)
130013        {
130014          if (fd[fdn].file != NULL)
130015            {
130016              k_printf ("%2i ", fdn);
130017              print_fd (fd);
130018            }
130019        }
130020  }
```

kernel/diag/print_file_head.c

«

Si veda la sezione u0.2.

```
140001  #include <sys/os16.h>
140002  #include <kernel/diag.h>
140003  #include <kernel/k_libc.h>
140004  //-----------------------------------------------------------------------
140005  void
140006  print_file_head (void)
140007  {
140008
140009      k_printf ("n. ref flg  mode uid  size lnks  dev ino\n");
140010  }
```

kernel/diag/print_file_list.c

«

Si veda la sezione u0.2.

```
150001  #include <sys/os16.h>
150002  #include <kernel/diag.h>
150003  #include <kernel/k_libc.h>
150004  //-----------------------------------------------------------------------
150005  void
150006  print_file_list (void)
150007  {
150008      int     fno;
150009      file_t *file = file_reference (0);
150010      //
150011      print_file_head ();
150012      //
150013      for (fno = 0; fno < FILE_MAX_SLOTS; fno++)
150014        {
150015          if (file[fno].references > 0)
150016            {
150017              print_file_num (fno);
150018            }
150019        }
150020  }
```

kernel/diag/print_file_num.c

«

Si veda la sezione u0.2.

```
160001  #include <sys/os16.h>
160002  #include <kernel/diag.h>
160003  #include <kernel/k_libc.h>
160004  #include <fcntl.h>
160005  //-----------------------------------------------------------------------
160006  void
160007  print_file_num (int fno)
160008  {
160009      file_t *file = file_reference (fno);
160010
160011      k_printf ("%2i %3i %c/%c %05o %3i %5li %4i %04x %3i",
160012                (unsigned int)      fno,
160013                (unsigned int)      file->references,
160014                (file->oflags & O_RDONLY ? 'r' : ' '),
160015                (file->oflags & O_WRONLY ? 'w' : ' '),
160016                (unsigned int)      file->inode->mode,
160017                (unsigned int)      file->inode->uid,
160018                (unsigned long int) file->inode->size,
160019                (unsigned int)      file->inode->links,
160020                (unsigned int)      file->inode->sb->device,
160021                (unsigned int)      file->inode->ino);
160022      k_printf ("\n");
160023  }
```



kernel/diag/print_hex_16.c

«

Si veda la sezione u0.2.

```
170001  #include <sys/os16.h>
170002  #include <kernel/diag.h>
170003  #include <kernel/k_libc.h>
170004  #include <inttypes.h>
170005  #include <stdio.h>
170006  //-----------------------------------------------------------------------
170007  void
170008  print_hex_16 (void *data, size_t elements)
170009  {
170010      uint16_t *element = (uint16_t *) data;
170011      int i;
170012      for (i = 0; i < elements; i++)
170013        {
170014          k_printf ("%04" PRIx16, (uint16_t) element[i]);
170015        }
170016  }
```

kernel/diag/print_hex_16_reverse.c

«

Si veda la sezione u0.2.

```
180001  #include <sys/os16.h>
180002  #include <kernel/diag.h>
180003  #include <kernel/k_libc.h>
180004  #include <inttypes.h>
180005  #include <stdio.h>
180006  //-----------------------------------------------------------------------
180007  void
180008  print_hex_16_reverse (void *data, size_t elements)
180009  {
180010      uint16_t *element = (uint16_t *) data;
180011      int i;
180012      for (i = 0; i < elements; i++)
180013        {
180014          k_printf ("%04" PRIx16, reverse_16_bit (element[i]));
180015        }
180016  }
```

kernel/diag/print_hex_32.c

«

Si veda la sezione u0.2.

```
190001  #include <sys/os16.h>
190002  #include <kernel/diag.h>
190003  #include <kernel/k_libc.h>
190004  #include <inttypes.h>
190005  #include <stdio.h>
190006  //-----------------------------------------------------------------------
190007  void
190008  print_hex_32 (void *data, size_t elements)
190009  {
190010      uint32_t *element = (uint32_t *) data;
190011      int i;
190012      for (i = 0; i < elements; i++)
190013        {
190014          k_printf ("%08" PRIx32, (uint32_t) element[i]);
190015        }
190016  }
```

kernel/diag/print_hex_32_reverse.c

«

Si veda la sezione u0.2.

```
200001  #include <sys/os16.h>
200002  #include <kernel/diag.h>
200003  #include <kernel/k_libc.h>
200004  #include <inttypes.h>
200005  #include <stdio.h>
200006  //-----------------------------------------------------------------------
200007  void
200008  print_hex_32_reverse (void *data, size_t elements)
200009  {
200010      uint32_t *element = (uint32_t *) data;
200011      int i;
200012      for (i = 0; i < elements; i++)
200013        {
200014          k_printf ("%08" PRIx32, reverse_32_bit (element[i]));
200015        }
200016  }
```

kernel/diag/print_hex_8.c

«

Si veda la sezione u0.2.

```
210001  #include <sys/os16.h>
210002  #include <kernel/diag.h>
210003  #include <kernel/k_libc.h>
210004  #include <inttypes.h>
210005  #include <stdio.h>
210006  //-----------------------------------------------------------------------
210007  void
210008  print_hex_8 (void *data, size_t elements)
210009  {
210010      uint8_t *element = (uint8_t *) data;
```

```
210011      int i;
210012      for (i = 0; i < elements; i++)
210013        {
210014          k_printf ("%02" PRIx8, (uint16_t) element[i]);
210015        }
210016  }
```

kernel/diag/print_hex_8_reverse.c

«

Si veda la sezione u0.2.

```
220001  #include <sys/os16.h>
220002  #include <kernel/diag.h>
220003  #include <kernel/k_libc.h>
220004  #include <inttypes.h>
220005  #include <stdio.h>
220006  //---------------------------------------------------------------------
220007  void
220008  print_hex_8_reverse (void *data, size_t elements)
220009  {
220010      uint8_t *element = (uint8_t *) data;
220011      int i;
220012      for (i = 0; i < elements; i++)
220013        {
220014          k_printf ("%02" PRIx8, reverse_8_bit (element[i]));
220015        }
220016  }
```

kernel/diag/print_inode.c

«

Si veda la sezione u0.2.

```
230001  #include <sys/os16.h>
230002  #include <kernel/diag.h>
230003  #include <kernel/k_libc.h>
230004  //---------------------------------------------------------------------
230005  void
230006  print_inode (inode_t *inode)
230007  {
230008      unsigned long int seconds;
230009      unsigned long int seconds_d;
230010      unsigned long int seconds_h;
230011      unsigned int      d;
230012      unsigned int      h;
230013      unsigned int      m;
230014      unsigned int      s;
230015      dev_t             device_attached = 0;
230016      //
230017      if (inode == NULL)
230018        {
230019          return;
230020        }
230021      //
230022      seconds    = inode->time;
230023      d          = seconds / 86400L;              // 24 * 60 * 60
230024      seconds_d  = d;
230025      seconds_d *= 86400;
230026      seconds   -= seconds_d;
230027      h          = seconds / 3840;                // 60 * 60
230028      seconds_h  = h;
230029      seconds_h *= 3840;
230030      seconds   -= seconds_h;
230031      m          = seconds / 60;
230032      s          = seconds % 60;
230033      //
230034      if (inode->sb_attached != NULL)
230035        {
230036          device_attached = inode->sb_attached->device;
230037        }
230038      //
230039      k_printf ("%04x %4i %3i %c %4x %06o %4i %3i %7li ",
230040                (unsigned int) inode->sb->device,
230041                (unsigned int) inode->ino,
230042                (unsigned int) inode->references,
230043                (inode->changed ? '!' : ' '),
230044                (unsigned int) device_attached,
230045                (unsigned int) inode->mode,
230046                (unsigned int) inode->uid,
230047                (unsigned int) inode->gid,
230048                (unsigned long int) inode->size);
230049
230050      k_printf ("%5i %2i:%02i:%02i %3i\n",
230051                d, h, m, s,
230052                (unsigned int) inode->links);
230053
230054  }
```

kernel/diag/print_inode_head.c

«

Si veda la sezione u0.2.

```
240001  #include <sys/os16.h>
240002  #include <kernel/diag.h>
240003  #include <kernel/k_libc.h>
240004  //---------------------------------------------------------------------
240005  void
240006  print_inode_head (void)
240007  {
```



```
240008      k_printf (" dev  ino ref c mntd   mode  uid gid    ");
240009      k_printf ("size  date     time lnk \n");
240010  }
```

kernel/diag/print_inode_list.c

«

Si veda la sezione u0.2.

```
250001  #include <sys/os16.h>
250002  #include <kernel/diag.h>
250003  #include <kernel/k_libc.h>
250004  //---------------------------------------------------------------------
250005  void
250006  print_inode_list (void)
250007  {
250008      ino_t ino;
250009      inode_t *inode = inode_reference (0, 0);
250010      print_inode_head ();
250011      for (ino = 0; ino < INODE_MAX_SLOTS; ino++)
250012        {
250013          if (inode[ino].references > 0)
250014            {
250015              print_inode (&inode[ino]);
250016            }
250017        }
250018  }
```

kernel/diag/print_inode_map.c

«

Si veda la sezione u0.2.

```
260001  #include <sys/os16.h>
260002  #include <kernel/diag.h>
260003  #include <kernel/k_libc.h>
260004  //---------------------------------------------------------------------
260005  void
260006  print_inode_map (sb_t *sb, uint16_t *bitmap)
260007  {
260008      size_t size;
260009      if (sb->inodes % 16)
260010        {
260011          size = sb->inodes/16 + 1;
260012        }
260013      else
260014        {
260015          size = sb->inodes/16;
260016        }
260017      print_hex_16_reverse (bitmap, size);
260018  }
```

kernel/diag/print_inode_zone_list.c

«

Si veda la sezione u0.2.

```
270001  #include <sys/os16.h>
270002  #include <kernel/diag.h>
270003  #include <kernel/k_libc.h>
270004  //---------------------------------------------------------------------
270005  void
270006  print_inode_zones_list (void)
270007  {
270008      ino_t ino;
270009      inode_t *inode = inode_reference (0, 0);
270010      print_inode_zones_head ();
270011      for (ino = 0; ino < INODE_MAX_SLOTS; ino++)
270012        {
270013          if (inode[ino].references > 0)
270014            {
270015              print_inode_zones (&inode[ino]);
270016            }
270017        }
270018  }
```

kernel/diag/print_inode_zones.c

«

Si veda la sezione u0.2.

```
280001  #include <sys/os16.h>
280002  #include <kernel/diag.h>
280003  #include <kernel/k_libc.h>
280004  //---------------------------------------------------------------------
280005  void
280006  print_inode_zones (inode_t *inode)
280007  {
280008      int i;
280009      //
280010      if (inode == NULL)
280011        {
280012          return;
280013        }
280014      //
280015      k_printf ("%04x %4i ",
280016                (unsigned int) inode->sb->device,
280017                (unsigned int) inode->ino);
280018
280019      for (i = 0; i < 7; i++)
280020        {
```

```
280021          if (inode->direct[i] != 0)
280022            {
280023              k_printf ("%04x ", (unsigned int) inode->direct[i]);
280024            }
280025          else
280026            {
280027              k_printf ("     ");
280028            }
280029        }
280030      if (inode->indirect1 != 0)
280031        {
280032          k_printf ("%04x ",     (unsigned int) inode->indirect1);
280033        }
280034      else
280035        {
280036          k_printf ("     ");
280037        }
280038      if (inode->indirect2 != 0)
280039        {
280040          k_printf ("%04x",      (unsigned int) inode->indirect2);
280041        }
280042      else
280043        {
280044          k_printf ("    ");
280045        }
280046      k_printf ("\n");
280047  }
```

## kernel/diag/print_inode_zones_head.c

«

Si veda la sezione u0.2.

```
290001  #include <sys/os16.h>
290002  #include <kernel/diag.h>
290003  #include <kernel/k_libc.h>
290004  //---------------------------------------------------------------------
290005  void
290006  print_inode_zones_head (void)
290007  {
290008      k_printf (" dev  ino zn_0 zn_1 zn_2 zn_3 zn_4 zn_5 zn_6 ");
290009      k_printf ("ind1 ind2\n");
290010  }
```

## kernel/diag/print_kmem.c

«

Si veda la sezione u0.2.

```
300001  #include <sys/os16.h>
300002  #include <kernel/diag.h>
300003  #include <kernel/k_libc.h>
300004  //---------------------------------------------------------------------
300005  extern uint16_t _ksp;
300006  extern uint16_t _etext;
300007  extern uint16_t _edata;
300008  extern uint16_t _end;
300009  //---------------------------------------------------------------------
300010  void
300011  print_kmem (void)
300012  {
300013      k_printf ("etext=%04x edata=%04x ebss=%04x ksp=%04x",
300014                (unsigned int) &_etext,
300015                (unsigned int) &_edata, (unsigned int) &_end, _ksp);
300016  }
```

## kernel/diag/print_mb_map.c

«

Si veda la sezione u0.2.

```
310001  #include <sys/os16.h>
310002  #include <kernel/diag.h>
310003  #include <kernel/k_libc.h>
310004  //---------------------------------------------------------------------
310005  void
310006  print_mb_map (void)
310007  {
310008      uint16_t    *mb = mb_reference ();
310009      unsigned int i;
310010      for (i = 0; i < (MEM_MAX_BLOCKS / 16); i++)
310011        {
310012          k_printf ("%04x", mb[i]);
310013        }
310014  }
```

## kernel/diag/print_memory_map.c

«

Si veda la sezione u0.2.

```
320001  #include <sys/os16.h>
320002  #include <kernel/diag.h>
320003  #include <kernel/k_libc.h>
320004  //---------------------------------------------------------------------
320005  void
320006  print_memory_map (void)
320007  {
320008      uint16_t    *mem_block[MEM_BLOCK_SIZE/2];
320009      uint16_t     block_rank;
320010      unsigned int b;
```



```
320011      unsigned int m;
320012      unsigned int i;
320013      addr_t       start;
320014
320015      start = 0;
320016      dev_io ((pid_t) -1, DEV_MEM, DEV_READ, start, mem_block,
320017              MEM_BLOCK_SIZE, NULL);
320018
320019      for (m = 0; m < MEM_MAX_BLOCKS; m++)
320020        {
320021          i = m % 16;
320022          if (i == 0)
320023            {
320024              block_rank = 0;
320025            }
320026          //
320027          for (b = 0; b < (MEM_BLOCK_SIZE / 2); b++)
320028            {
320029              if (mem_block[b])
320030                {
320031                  block_rank |= (0x8000 >> i);
320032                  break;
320033                }
320034            }
320035          //
320036          if (i == 15)
320037            {
320038              k_printf ("%04x", block_rank);
320039            }
320040          //
320041          start += MEM_BLOCK_SIZE;
320042          dev_io ((pid_t) -1, DEV_MEM, DEV_READ, start, mem_block,
320043                  MEM_BLOCK_SIZE, NULL);
320044        }
320045  }
```

## kernel/diag/print_proc_head.c

«

Si veda la sezione u0.2.

```
330001  #include <sys/os16.h>
330002  #include <kernel/diag.h>
330003  #include <kernel/k_libc.h>
330004  //---------------------------------------------------------------------
330005  void
330006  print_proc_head (void)
330007  {
330008      k_printf (
330009  "pp  p pg                                                    \n"
330010  "id id rp  tty  uid euid suid usage s iaddr isiz daddr dsiz sp   name\n"
330011            );
330012  }
```

## kernel/diag/print_proc_list.c

«

Si veda la sezione u0.2.

```
340001  #include <sys/os16.h>
340002  #include <kernel/diag.h>
340003  #include <kernel/k_libc.h>
340004  //---------------------------------------------------------------------
340005  void
340006  print_proc_list (void)
340007  {
340008      pid_t   pid;
340009      proc_t *ps;
340010      //
340011      print_proc_head ();
340012      //
340013      for (pid = 0; pid < PROCESS_MAX; pid++)
340014        {
340015          ps = proc_reference (pid);
340016          if (ps != NULL && ps->status > 0)
340017            {
340018              print_proc_pid (ps, pid);
340019            }
340020        }
340021  }
```

## kernel/diag/print_proc_pid.c

«

Si veda la sezione u0.2.

```
350001  #include <sys/os16.h>
350002  #include <kernel/diag.h>
350003  #include <kernel/k_libc.h>
350004  //---------------------------------------------------------------------
350005  void
350006  print_proc_pid (proc_t *ps, pid_t pid)
350007  {
350008      char  stat;
350009      switch (ps->status)
350010        {
350011          case PROC_EMPTY   : stat = '-'; break;
350012          case PROC_CREATED : stat = 'c'; break;
350013          case PROC_READY   : stat = 'r'; break;
350014          case PROC_RUNNING : stat = 'R'; break;
350015          case PROC_SLEEPING: stat = 's'; break;
```

```
350016          case PROC_ZOMBIE  : stat = 'z'; break;
350017          default           : stat = '?'; break;
350018        }
350019
350020      k_printf ("%2i %2i %2i %04x %4i %4i %4i %02i.%02i %c %05lx %04x ",
350021                (unsigned int) ps->ppid,
350022                (unsigned int) pid,
350023                (unsigned int) ps->pgrp,
350024                (unsigned int) ps->device_tty,
350025                (unsigned int) ps->uid,
350026                (unsigned int) ps->euid,
350027                (unsigned int) ps->suid,
350028                (unsigned int) ((ps->usage / CLOCKS_PER_SEC) / 60),
350029                (unsigned int) ((ps->usage / CLOCKS_PER_SEC) % 60),
350030                stat,
350031                (unsigned long int) ps->address_i,
350032                (unsigned int) ps->size_i);
350033
350034      k_printf ("%05lx %04x %04x %s",
350035                (unsigned long int) ps->address_d,
350036                (unsigned int) ps->size_d,
350037                (unsigned int) ps->sp,
350038                               ps->name);
350039
350040      k_printf ("\n");
350041  }
```

## kernel/diag/print_segments.c

«

Si veda la sezione u0.2.

```
360001  #include <sys/os16.h>
360002  #include <kernel/diag.h>
360003  #include <kernel/k_libc.h>
360004  //-----------------------------------------------------------------------
360005  void
360006  print_segments (void)
360007  {
360008      k_printf ("CS=%04x DS=%04x SS=%04x ES=%04x BP=%04x SP=%04x ",
360009                cs (), ds (), ss (), es (), bp (), sp ());
360010      k_printf ("heap_min=%04x", heap_min ());
360011  }
```

## kernel/diag/print_superblock.c

«

Si veda la sezione u0.2.

```
370001  #include <sys/os16.h>
370002  #include <kernel/diag.h>
370003  #include <kernel/k_libc.h>
370004  //-----------------------------------------------------------------------
370005  void
370006  print_superblock (sb_t *sb)
370007  {
370008      k_printf ("Inodes:           %i\n", sb->inodes);
370009      k_printf ("Blocks:           %i\n", sb->zones);
370010      k_printf ("First data zone:  %i\n", sb->first_data_zone);
370011      k_printf ("Zone size:        %i\n", (1024 << sb->log2_size_zone));
370012      k_printf ("Max file size:    %li\n", sb->max_file_size);
370013      k_printf ("Inode map blocks: %i\n", sb->map_inode_blocks);
370014      k_printf ("Zone map blocks:  %i\n", sb->map_zone_blocks);
370015  }
```

## kernel/diag/print_time.c

«

Si veda la sezione u0.2.

```
380001  #include <sys/os16.h>
380002  #include <kernel/diag.h>
380003  #include <kernel/k_libc.h>
380004  //-----------------------------------------------------------------------
380005  void
380006  print_time (void)
380007  {
380008      unsigned long int ticks   = k_clock ();
380009      unsigned long int seconds = k_time (NULL);
380010      unsigned int      h       = seconds / 60 / 60;
380011      unsigned int      m       = seconds / 60 - h * 60;
380012      unsigned int      s       = seconds - m * 60 - h * 60 * 60;
380013      k_printf ("clock=%08lx, time elapsed=%02u:%02u:%02u",
380014                ticks, h, m, s);
380015  }
```

## kernel/diag/print_zone_map.c

«

Si veda la sezione u0.2.

```
390001  #include <sys/os16.h>
390002  #include <kernel/diag.h>
390003  #include <kernel/k_libc.h>
390004  //-----------------------------------------------------------------------
390005  void
390006  print_zone_map (sb_t *sb, uint16_t *bitmap)
390007  {
390008      size_t       size;
390009      unsigned int data_zones = sb->zones - sb->first_data_zone;
390010      if (data_zones % 16)
```



```
390011        {
390012          size = data_zones/16 + 1;
390013        }
390014      else
390015        {
390016          size = data_zones/16;
390017        }
390018      print_hex_16_reverse (bitmap, size);
390019  }
```

## kernel/diag/reverse_16_bit.c

«

Si veda la sezione u0.2.

```
400001  #include <sys/os16.h>
400002  #include <kernel/diag.h>
400003  #include <kernel/k_libc.h>
400004  #include <inttypes.h>
400005  //-----------------------------------------------------------------------
400006  uint16_t
400007  reverse_16_bit (uint16_t source)
400008  {
400009      uint16_t destination = 0;
400010      uint16_t mask_src;
400011      uint16_t mask_dst;
400012      int      i;
400013      for (i = 0; i < 16; i++)
400014        {
400015          mask_src = 0x0001 << i;
400016          mask_dst = 0x8000 >> i;
400017          if (source & mask_src)
400018            {
400019              destination |= mask_dst;
400020            }
400021        }
400022      return (destination);
400023  }
```

## kernel/diag/reverse_32_bit.c

«

Si veda la sezione u0.2.

```
410001  #include <sys/os16.h>
410002  #include <kernel/diag.h>
410003  #include <kernel/k_libc.h>
410004  //-----------------------------------------------------------------------
410005  uint32_t
410006  reverse_32_bit (uint32_t source)
410007  {
410008      uint32_t destination = 0;
410009      uint32_t mask_src;
410010      uint32_t mask_dst;
410011      int      i;
410012      for (i = 0; i < 32; i++)
410013        {
410014          mask_src = 0x00000001 << i;
410015          mask_dst = 0x80000000 >> i;
410016          if (source & mask_src)
410017            {
410018              destination |= mask_dst;
410019            }
410020        }
410021      return (destination);
410022  }
```

## kernel/diag/reverse_8_bit.c

«

Si veda la sezione u0.2.

```
420001  #include <sys/os16.h>
420002  #include <kernel/diag.h>
420003  #include <kernel/k_libc.h>
420004  #include <inttypes.h>
420005  //-----------------------------------------------------------------------
420006  uint8_t
420007  reverse_8_bit (uint8_t source)
420008  {
420009      uint8_t destination = 0;
420010      uint8_t mask_src;
420011      uint8_t mask_dst;
420012      int      i;
420013      for (i = 0; i < 8; i++)
420014        {
420015          mask_src = 0x01 << i;
420016          mask_dst = 0x80 >> i;
420017          if (source & mask_src)
420018            {
420019              destination |= mask_dst;
420020            }
420021        }
420022      return (destination);
420023  }
```

«

Si veda la sezione u0.3.

```
430001  #ifndef _KERNEL_FS_H
430002  #define _KERNEL_FS_H 1
430003
430004  #include <stdint.h>
430005  #include <sys/types.h>
430006  #include <kernel/memory.h>
430007  #include <sys/os16.h>
430008  #include <sys/stat.h>
430009  #include <stdint.h>
430010  #include <const.h>
430011  #include <stdio.h>
430012  #include <limits.h>
430013
430014  //------------------------------------------------------------------------
430015  #define SB_MAX_INODE_BLOCKS          1  // 8192 inodes max.
430016  #define SB_MAX_ZONE_BLOCKS           1  // 8192 data-zones max.
430017  #define SB_BLOCK_SIZE             1024  // Fixed for Minix file system.
430018  #define SB_MAX_ZONE_SIZE          2048  // log2 max is 1.
430019  #define SB_MAP_INODE_SIZE         (SB_MAX_INODE_BLOCKS*512) // [1]
430020  #define SB_MAP_ZONE_SIZE          (SB_MAX_ZONE_BLOCKS*512)  // [1]
430021  //
430022  // [1] blocks * (1024 * 8 / 16) = number of bits, divided 16.
430023  //
430024  //------------------------------------------------------------------------
430025  #define INODE_MAX_INDIRECT_ZONES  (SB_MAX_ZONE_SIZE/2)      // [2]
430026
430027  #define INODE_MAX_REFERENCES      0xFF
430028  //
430029  // [2] number of zone pointers contained inside a zone, used
430030  //     as an indirect inode list (a pointer = 16 bits = 2 bytes).
430031  //
430032  //------------------------------------------------------------------------
430033  typedef uint16_t        zno_t;          // Zone number.
430034  //------------------------------------------------------------------------
430035  // The structured type 'inode_t' must be pre-declared here, because
430036  // the type sb_t, described before the inode structure, has a member
430037  // pointing to a type 'inode_t'. So, must be declared previously
430038  // the type 'inode_t' as made of a type 'struct inode', then the
430039  // structure 'inode' can be described. But for a matter of coherence,
430040  // all other structured data declared inside this file follow the
430041  // same procedure.
430042  //
430043  typedef struct sb        sb_t;
430044  typedef struct inode     inode_t;
430045  typedef struct file      file_t;
430046  typedef struct fd        fd_t;
430047  typedef struct directory directory_t;
430048  //------------------------------------------------------------------------
430049  #define SB_MAX_SLOTS  2                 // Handle max 2 file systems.
430050
430051  struct sb {                             // File system super block:
430052      uint16_t      inodes;               // inodes available;
430053      uint16_t      zones;                // zones available (disk size);
430054      uint16_t      map_inode_blocks;     // inode bit map blocks;
430055      uint16_t      map_zone_blocks;      // data-zone bit map blocks;
430056      uint16_t      first_data_zone;      // first data-zone;
430057      uint16_t      log2_size_zone;       // log_2 (size_zone/block_size);
430058      uint32_t      max_file_size;        // max file size in bytes;
430059      uint16_t      magic_number;         // file system magic number.
430060      //--------------------------------------------------------------
430061      // Extra management data, not saved inside the file system
430062      // super block.
430063      //--------------------------------------------------------------
430064      dev_t         device;                       // FS device [3]
430065      inode_t      *inode_mounted_on;             // [4]
430066      blksize_t     blksize;                      // Calculated zone size.
430067      int           options;                      // [5]
430068      uint16_t      map_inode[SB_MAP_INODE_SIZE];
430069      uint16_t      map_zone[SB_MAP_ZONE_SIZE];
430070      char          changed;
430071  };
430072
430073  extern sb_t    sb_table[SB_MAX_SLOTS];
430074  //
430075  // [3] the member 'device' must be kept at the same position, because
430076  //     it is used to calculate the super block header size, saved on
430077  //     disk.
430078  //
430079  // [4] If this pointer is not NULL, the super block is related to a
430080  //     device mounted on a directory. The inode of such directory is
430081  //     recorded here. Please note that it is type 'void *', instead of
430082  //     type 'inode_t', because type 'inode_t' is declared after type
430083  //     'sb_t'.
430084  //     Please note that the type 'sb_t' is declared before the
430085  //     type 'inode_t', but this member points to a type 'inode_t'.
430086  //     This is the reason because it was necessary to declare first
430087  //     the type 'inode_t' as made of 'struct inode', to be described
430088  //     later. For coherence, all derived type made of structured data,
430089  //     are first declared as structure, and then, later, described.
430090  //
430091  // [5] Mount options can be only 'MOUNT_DEFAULT' or 'MOUNT_RO',
430092  //     as defined inside file 'lib/sys/os16.h'.
430093  //
430094  //------------------------------------------------------------------------
430095  #define INODE_MAX_SLOTS           (8 * OPEN_MAX)
430096
430097  struct inode {                          // Inode (32 byte total):
430098      mode_t        mode;                 // file type and permissions;
```



```
430099      uid_t         uid;                  // user ID (16 bit);
430100      ssize_t       size;                 // file size in bytes;
430101      time_t        time;                 // file data modification time;
430102      uint8_t       gid;                  // group ID (8 bit);
430103      uint8_t       links;                // links to the inode;
430104      zno_t         direct[7];            // direct zones;
430105      zno_t         indirect1;            // indirect zones;
430106      zno_t         indirect2;            // double indirect zones.
430107      //--------------------------------------------------------------
430108      // Extra management data, not saved inside the disk file system.
430109      //--------------------------------------------------------------
430110      sb_t         *sb;                   // Inode's super block. [7]
430111      ino_t         ino;                  // Inode number.
430112      sb_t         *sb_attached;          // [8]
430113      blkcnt_t      blkcnt;               // Rounded size/blksize.
430114      unsigned char references;           // Run time active references.
430115      char          changed;              // 1 == to be saved.
430116  };
430117
430118  extern inode_t inode_table[INODE_MAX_SLOTS];
430119  //
430120  // [7] the member 'sb' must be kept at the same position, because
430121  //     it is used to calculate the inode header size, saved on disk.
430122  //
430123  // [8] If the inode is a mount point for another device, the other
430124  //     super block pointer is saved inside 'sb_attached'.
430125  //
430126  //------------------------------------------------------------------------
430127  #define FILE_MAX_SLOTS             (16 * OPEN_MAX)
430128
430129  struct file {
430130      int           references;
430131      off_t         offset;               // File position.
430132      int           oflags;               // Open mode: r/w/r+w [9]
430133      inode_t      *inode;
430134  };
430135
430136  extern file_t file_table[FILE_MAX_SLOTS];
430137  //
430138  // [9] the member 'oflags' can get only O_RDONLY, O_WRONLY, O_RDWR,
430139  //     (from header 'fcntl.h') combined with OR binary operator.
430140  //
430141  //------------------------------------------------------------------------
430142  struct fd {
430143      int           fl_flags;             // File status flags and file
430144                                          // access modes. [10]
430145      int           fd_flags;             // File descriptor flags:
430146                                          // currently only FD_CLOEXEC.
430147      file_t       *file;                 // Pointer to the file table.
430148  };
430149  //
430150  // [10] the member 'fl_flags' can get only O_RDONLY, O_WRONLY, O_RDWR,
430151  //      O_CREAT, O_EXCL, O_NOCTTY, O_TRUNC and O_APPEND
430152  //      (from header 'fcntl.h') combined with OR binary
430153  //      operator. Options like O_DSYNC, O_NONBLOCK, O_RSYNC and O_SYNC
430154  //      are not taken into consideration by os16.
430155  //
430156  //------------------------------------------------------------------------
430157  struct directory {                      // Directory entry:
430158      ino_t         ino;                  // inode number;
430159      char          name[NAME_MAX];       // file name.
430160  };
430161  //------------------------------------------------------------------------
430162  sb_t    *sb_reference    (dev_t device);
430163  sb_t    *sb_mount        (dev_t device, inode_t **inode_mnt,
430164                            int options);
430165  sb_t    *sb_get          (dev_t device, sb_t *sb);
430166  int      sb_save         (sb_t *sb);
430167  int      sb_zone_status  (sb_t *sb, zno_t zone);
430168  int      sb_inode_status (sb_t *sb, ino_t ino);
430169  //------------------------------------------------------------------------
430170  zno_t    zone_alloc           (sb_t *sb);
430171  int      zone_free            (sb_t *sb, zno_t zone);
430172  int      zone_read            (sb_t *sb, zno_t zone, void *buffer);
430173  int      zone_write           (sb_t *sb, zno_t zone, void *buffer);
430174  //------------------------------------------------------------------------
430175  inode_t *inode_reference      (dev_t device, ino_t ino);
430176  inode_t *inode_alloc          (dev_t device, mode_t mode, uid_t uid);
430177  int      inode_free           (inode_t *inode);
430178  inode_t *inode_get            (dev_t device, ino_t ino);
430179  int      inode_save           (inode_t *inode);
430180  int      inode_put            (inode_t *inode);
430181  int      inode_truncate       (inode_t *inode);
430182  zno_t    inode_zone           (inode_t *inode, zno_t fzone, int write);
430183  inode_t *inode_stdio_dev_make (dev_t device, mode_t mode);
430184  blkcnt_t inode_fzones_read    (inode_t *inode, zno_t zone_start,
430185                                 void *buffer, blkcnt_t blkcnt);
430186  blkcnt_t inode_fzones_write   (inode_t *inode, zno_t zone_start,
430187                                 void *buffer, blkcnt_t blkcnt);
430188  ssize_t  inode_file_read      (inode_t *inode, off_t offset,
430189                                 void *buffer, size_t count, int *eof);
430190  ssize_t  inode_file_write     (inode_t *inode, off_t offset,
430191                                 void *buffer, size_t count);
430192  int      inode_check          (inode_t *inode, mode_t type,
430193                                 int perm, uid_t uid);
430194  int      inode_dir_empty      (inode_t *inode);
430195  //------------------------------------------------------------------------
430196  file_t  *file_reference       (int fno);
430197  file_t  *file_stdio_dev_make  (dev_t device, mode_t mode, int oflags);
430198  //------------------------------------------------------------------------
430199  inode_t *path_inode      (pid_t pid, const char *path);
```

```
430200 int      path_chdir      (pid_t pid, const char *path);
430201 dev_t    path_device     (pid_t pid, const char *path);
430202 int      path_full       (const char *path,
430203                           const char *path_cwd,
430204                           char *full_path);
430205 int      path_fix        (char *path);
430206 inode_t *path_inode_link (pid_t pid, const char *path, inode_t *inode,
430207                           mode_t mode);
430208 int      path_link       (pid_t pid, const char *path_old,
430209                           const char *path_new);
430210 int      path_mkdir      (pid_t pid, const char *path, mode_t mode);
430211 int      path_mknod      (pid_t pid, const char *path, mode_t mode,
430212                           dev_t device);
430213 int      path_mount      (pid_t pid, const char *path_dev,
430214                           const char *path_mnt,
430215                           int options);
430216 int      path_umount     (pid_t pid, const char *path_mnt);
430217 int      path_stat       (pid_t pid, const char *path,
430218                           struct stat *buffer);
430219 int      path_chmod      (pid_t pid, const char *path, mode_t mode);
430220 int      path_chown      (pid_t pid, const char *path, uid_t uid,
430221                           gid_t gid);
430222 int      path_unlink     (pid_t pid, const char *path);
430223 //-------------------------------------------------------------------------
430224 fd_t    *fd_reference (pid_t pid, int *fdn);
430225 int      fd_chmod     (pid_t pid, int fdn, mode_t mode);
430226 int      fd_chown     (pid_t pid, int fdn, uid_t uid, gid_t gid);
430227 int      fd_close     (pid_t pid, int fdn);
430228 int      fd_fcntl     (pid_t pid, int fdn, int cmd, int arg);
430229 int      fd_dup       (pid_t pid, int fdn_old, int fdn_min);
430230 int      fd_dup2      (pid_t pid, int fdn_old, int fdn_new);
430231 off_t    fd_lseek     (pid_t pid, int fdn, off_t offset, int whence);
430232 int      fd_open      (pid_t pid, const char *path, int oflags,
430233                        mode_t mode);
430234 ssize_t  fd_read      (pid_t pid, int fdn, void *buffer, size_t count,
430235                        int *eof);
430236 int      fd_stat      (pid_t pid, int fdn, struct stat *buffer);
430237 ssize_t  fd_write     (pid_t pid, int fdn, const void *buffer,
430238                        size_t count);
430239 //-------------------------------------------------------------------------
430240
430241 #endif
```

## kernel/fs/fd_chmod.c

«

Si veda la sezione i159.3.1.

```
440001 #include <kernel/proc.h>
440002 #include <kernel/k_libc.h>
440003 #include <sys/stat.h>
440004 #include <errno.h>
440005 //-------------------------------------------------------------------------
440006 int
440007 fd_chmod (pid_t pid, int fdn, mode_t mode)
440008 {
440009     proc_t   *ps;
440010     inode_t  *inode;
440011     //
440012     // Get process.
440013     //
440014     ps = proc_reference (pid);
440015     //
440016     // Verify if the file descriptor is valid.
440017     //
440018     if (ps->fd[fdn].file == NULL)
440019       {
440020         errset (EBADF);         // Bad file descriptor.
440021         return (-1);
440022       }
440023     //
440024     // Reach the inode.
440025     //
440026     inode = ps->fd[fdn].file->inode;
440027     //
440028     // Verify to be the owner, or at least to be UID == 0.
440029     //
440030     if (ps->euid != inode->uid && ps->euid != 0)
440031       {
440032         errset (EACCES);        // Permission denied.
440033         return (-1);
440034       }
440035     //
440036     // Update the mode: the file type is kept and the
440037     // rest is taken form the parameter 'mode'.
440038     //
440039     inode->mode = (S_IFMT & inode->mode) | (~S_IFMT & mode);
440040     //
440041     // Save the inode.
440042     //
440043     inode->changed = 1;
440044     inode_save (inode);
440045     //
440046     // Return.
440047     //
440048     return (0);
440049 }
```



## kernel/fs/fd_chown.c

«

Si veda la sezione i159.3.2.

```
450001 #include <kernel/proc.h>
450002 #include <kernel/k_libc.h>
450003 #include <errno.h>
450004 //-------------------------------------------------------------------------
450005 int
450006 fd_chown (pid_t pid, int fdn, uid_t uid, gid_t gid)
450007 {
450008     proc_t   *ps;
450009     inode_t  *inode;
450010     //
450011     // Get process.
450012     //
450013     ps = proc_reference (pid);
450014     //
450015     // Verify if the file descriptor is valid.
450016     //
450017     if (ps->fd[fdn].file == NULL)
450018       {
450019         errset (EBADF);         // Bad file descriptor.
450020         return (-1);
450021       }
450022     //
450023     // Reach the inode.
450024     //
450025     inode = ps->fd[fdn].file->inode;
450026     //
450027     // Verify to be root, as the ability to change group
450028     // is not taken into consideration.
450029     //
450030     if (ps->euid != 0)
450031       {
450032         errset (EACCES);        // Permission denied.
450033         return (-1);
450034       }
450035     //
450036     // Update the ownership.
450037     //
450038     if (uid != -1)
450039       {
450040         inode->uid     = uid;
450041         inode->changed = 1;
450042       }
450043     if (gid != -1)
450044       {
450045         inode->gid     = gid;
450046         inode->changed = 1;
450047       }
450048     //
450049     // Save the inode.
450050     //
450051     inode->changed = 1;
450052     inode_save (inode);
450053     //
450054     // Return.
450055     //
450056     return (0);
450057 }
```

## kernel/fs/fd_close.c

«

Si veda la sezione i159.3.3.

```
460001 #include <kernel/proc.h>
460002 #include <kernel/k_libc.h>
460003 #include <errno.h>
460004 //-------------------------------------------------------------------------
460005 int
460006 fd_close (pid_t pid, int fdn)
460007 {
460008     inode_t           *inode;
460009     file_t            *file;
460010     fd_t *fd;
460011     //
460012     // Get file descriptor.
460013     //
460014     fd = fd_reference (pid, &fdn);
460015     if (fd == NULL                ||
460016         fd->file == NULL          ||
460017         fd->file->inode == NULL )
460018       {
460019         errset (EBADF);         // Bad file descriptor.
460020         return (-1);
460021       }
460022     //
460023     // Get file.
460024     //
460025     file = fd->file;
460026     //
460027     // Get inode.
460028     //
460029     inode = file->inode;
460030     //
460031     // Reduce references inside the file table item
460032     // and remove item if it reaches zero.
460033     //
460034     file->references--;
460035     if (file->references == 0)
```

```
460036        {
460037          file->oflags = 0;
460038          file->inode  = NULL;
460039          //
460040          // Put inode, because there are no more file references.
460041          //
460042          inode_put (inode);
460043        }
460044      //
460045      // Remove file descriptor.
460046      //
460047      fd->fl_flags = 0;
460048      fd->fd_flags = 0;
460049      fd->file  = NULL;
460050      //
460051      //
460052      //
460053      return (0);
460054  }
```

kernel/fs/fd_dup.c

«

Si veda la sezione i159.3.4.

```
470001  #include <kernel/proc.h>
470002  #include <kernel/k_libc.h>
470003  #include <errno.h>
470004  #include <fcntl.h>
470005  //-------------------------------------------------------------------------
470006  int
470007  fd_dup (pid_t pid, int fdn_old, int fdn_min)
470008  {
470009      proc_t *ps;
470010      int     fdn_new;
470011      //
470012      // Verify argument.
470013      //
470014      if (fdn_min < 0 || fdn_min >= OPEN_MAX)
470015        {
470016          errset (EINVAL);                 // Invalid argument.
470017          return (-1);
470018        }
470019      //
470020      // Get process.
470021      //
470022      ps = proc_reference (pid);
470023      //
470024      // Verify if 'fdn_old' is a valid value.
470025      //
470026      if (fdn_old < 0                     ||
470027          fdn_old >= OPEN_MAX             ||
470028          ps->fd[fdn_old].file == NULL)
470029        {
470030          errset (EBADF);                 // Bad file descriptor.
470031          return (-1);
470032        }
470033      //
470034      // Find the first free slot and duplicate the file descriptor.
470035      //
470036      for (fdn_new = fdn_min; fdn_new < OPEN_MAX; fdn_new++)
470037        {
470038          if (ps->fd[fdn_new].file == NULL)
470039            {
470040              ps->fd[fdn_new].fl_flags = ps->fd[fdn_old].fl_flags;
470041              ps->fd[fdn_new].fd_flags
470042                  = ps->fd[fdn_old].fd_flags & ~FD_CLOEXEC;
470043              ps->fd[fdn_new].file = ps->fd[fdn_old].file;
470044              ps->fd[fdn_new].file->references++;
470045              return (fdn_new);
470046            }
470047        }
470048      //
470049      // No fd slot available.
470050      //
470051      errset (EMFILE);            // Too many open files.
470052      return (-1);
470053  }
```

kernel/fs/fd_dup2.c

«

Si veda la sezione i159.3.4.

```
480001  #include <kernel/proc.h>
480002  #include <kernel/k_libc.h>
480003  #include <errno.h>
480004  #include <fcntl.h>
480005  //-------------------------------------------------------------------------
480006  int
480007  fd_dup2 (pid_t pid, int fdn_old, int fdn_new)
480008  {
480009      proc_t *ps;
480010      int     status;
480011      //
480012      // Get process.
480013      //
480014      ps = proc_reference (pid);
480015      //
480016      // Verify if 'fdn_old' is a valid value.
480017      //
480018      if (fdn_old < 0                     ||
```



```
480019          fdn_old >= OPEN_MAX             ||
480020          ps->fd[fdn_old].file == NULL)
480021        {
480022          errset (EBADF);                 // Bad file descriptor.
480023          return (-1);
480024        }
480025      //
480026      // Check if 'fd_old' and 'fd_new' are the same.
480027      //
480028      if (fdn_old == fdn_new)
480029        {
480030          return (fdn_new);
480031        }
480032      //
480033      // Close 'fdn_new' if it is open and copy 'fdn_old' into it.
480034      //
480035      if (ps->fd[fdn_new].file != NULL)
480036        {
480037          status = fd_close (pid, fdn_new);
480038          if (status != 0)
480039            {
480040              return (-1);
480041            }
480042        }
480043      ps->fd[fdn_new].fl_flags = ps->fd[fdn_old].fl_flags;
480044      ps->fd[fdn_new].fd_flags = ps->fd[fdn_old].fd_flags & ~FD_CLOEXEC;
480045      ps->fd[fdn_new].file = ps->fd[fdn_old].file;
480046      ps->fd[fdn_new].file->references++;
480047      return (fdn_new);
480048  }
```

kernel/fs/fd_fcntl.c

«

Si veda la sezione i159.3.6.

```
490001  #include <kernel/proc.h>
490002  #include <kernel/k_libc.h>
490003  #include <errno.h>
490004  #include <fcntl.h>
490005  //-------------------------------------------------------------------------
490006  int
490007  fd_fcntl (pid_t pid, int fdn, int cmd, int arg)
490008  {
490009      proc_t   *ps;
490010      inode_t  *inode;
490011      int       mask;
490012      //
490013      // Get process.
490014      //
490015      ps = proc_reference (pid);
490016      //
490017      // Verify if the file descriptor is valid.
490018      //
490019      if (ps->fd[fdn].file == NULL)
490020        {
490021          errset (EBADF);                 // Bad file descriptor.
490022          return (-1);
490023        }
490024      //
490025      // Reach the inode.
490026      //
490027      inode = ps->fd[fdn].file->inode;
490028      //
490029      //
490030      //
490031      switch (cmd)
490032        {
490033          case F_DUPFD:
490034              return (fd_dup (pid, fdn, arg));
490035              break;
490036          case F_GETFD:
490037              return (ps->fd[fdn].fd_flags);
490038              break;
490039          case F_SETFD:
490040              ps->fd[fdn].fd_flags = arg;
490041              return (0);
490042          case F_GETFL:
490043              return (ps->fd[fdn].fl_flags);
490044          case F_SETFL:
490045              //
490046              // Calculate a mask with bits that are not to be set.
490047              //
490048              mask = (O_ACCMODE
490049                    | O_CREAT
490050                    | O_EXCL
490051                    | O_NOCTTY
490052                    | O_TRUNC);
490053              //
490054              // Set to zero the bits that are not to be set from
490055              // the argument.
490056              //
490057              arg = (arg & ~mask);
490058              //
490059              // Set to zero the bit that *are* to be set.
490060              //
490061              ps->fd[fdn].fl_flags &= mask;
490062              //
490063              // Set the bits, already filtered inside the argument.
490064              //
490065              ps->fd[fdn].fl_flags |= arg;
490066              //
```

```
490067              return (0);
490068          default:
490069              errset (EINVAL);                    // Not implemented.
490070              return (-1);
490071          }
490072  }
```

## kernel/fs/fd_lseek.c

«

Si veda la sezione i159.3.7.

```
500001  #include <kernel/proc.h>
500002  #include <kernel/k_libc.h>
500003  #include <errno.h>
500004  //---------------------------------------------------------------------
500005  off_t
500006  fd_lseek (pid_t pid, int fdn, off_t offset, int whence)
500007  {
500008      inode_t           *inode;
500009      file_t            *file;
500010      fd_t *fd;
500011      off_t              test_offset;
500012      //
500013      // Get file descriptor.
500014      //
500015      fd = fd_reference (pid, &fdn);
500016      if (fd == NULL               ||
500017          fd->file == NULL         ||
500018          fd->file->inode == NULL )
500019        {
500020          errset (EBADF);                 // Bad file descriptor.
500021          return (-1);
500022        }
500023      //
500024      // Get file table item.
500025      //
500026      file = fd->file;
500027      //
500028      // Get inode.
500029      //
500030      inode = file->inode;
500031      //
500032      // Change position depending on the 'whence' parameter.
500033      //
500034      if (whence == SEEK_SET)
500035        {
500036          if (offset < 0)
500037            {
500038              errset (EINVAL);                    // Invalid argument.
500039              return ((off_t) -1);
500040            }
500041          else
500042            {
500043              fd->file->offset = offset;
500044            }
500045        }
500046      else if (whence == SEEK_CUR)
500047        {
500048          test_offset  = fd->file->offset;
500049          test_offset += offset;
500050          if (test_offset < 0)
500051            {
500052              errset (EINVAL);                    // Invalid argument.
500053              return ((off_t) -1);
500054            }
500055          else
500056            {
500057              fd->file->offset = test_offset;
500058            }
500059        }
500060      else if (whence == SEEK_END)
500061        {
500062          test_offset  = inode->size;
500063          test_offset += offset;
500064          if (test_offset < 0)
500065            {
500066              errset (EINVAL);                    // Invalid argument.
500067              return ((off_t) -1);
500068            }
500069          else
500070            {
500071              fd->file->offset = test_offset;
500072            }
500073        }
500074      else
500075        {
500076          errset (EINVAL);                        // Invalid argument.
500077          return ((off_t) -1);
500078        }
500079      //
500080      // Return the new file position.
500081      //
500082      return (fd->file->offset);
500083  }
```



## kernel/fs/fd_open.c

«

Si veda la sezione i159.3.8.

```
510001  #include <kernel/proc.h>
510002  #include <kernel/k_libc.h>
510003  #include <errno.h>
510004  #include <fcntl.h>
510005  //---------------------------------------------------------------------
510006  int
510007  fd_open (pid_t pid, const char *path, int oflags, mode_t mode)
510008  {
510009      proc_t   *ps;
510010      inode_t  *inode;
510011      int       status;
510012      file_t   *file;
510013      fd_t     *fd;
510014      int       fdn;
510015      char      full_path[PATH_MAX];
510016      int       perm;
510017      tty_t    *tty;
510018      mode_t    umask;
510019      int       errno_save;
510020      //
510021      // Get process.
510022      //
510023      ps = proc_reference (pid);
510024      //
510025      // Correct the mode with the umask. As it is not a directory, to the
510026      // mode are removed execution and sticky permissions.
510027      //
510028      umask = ps->umask | 01111;
510029      mode &= ~umask;
510030      //
510031      // Check open options.
510032      //
510033      if (oflags & O_WRONLY)
510034        {
510035          //
510036          // The file is to be opened for write, or for read/write.
510037          // Try to get inode.
510038          //
510039          inode = path_inode (pid, path);
510040          if (inode == NULL)
510041            {
510042              //
510043              // Cannot get the inode. See if there is the creation
510044              // option.
510045              //
510046              if (oflags & O_CREAT)
510047                {
510048                  //
510049                  // Try to create the missing inode: the file must be a
510050                  // regular one, so add the mode.
510051                  //
510052                  path_full (path, ps->path_cwd, full_path);
510053                  inode = path_inode_link (pid, full_path, NULL,
510054                                           (mode | S_IFREG));
510055                  if (inode == NULL)
510056                    {
510057                      //
510058                      // Sorry: cannot create the inode! Variable 'errno'
510059                      // is already set by 'path_inode_link()'.
510060                      //
510061                      errset (errno);
510062                      return (-1);
510063                    }
510064                }
510065              else
510066                {
510067                  //
510068                  // Cannot open the inode. Variable 'errno'
510069                  // should be already set by 'path_inode()'.
510070                  //
510071                  errset (errno);
510072                  return (-1);
510073                }
510074            }
510075          //
510076          // The inode was read or created: check if it must be
510077          // truncated. It can be truncated only if it is a regular
510078          // file.
510079          //
510080          if (oflags & O_TRUNC && inode->mode & S_IFREG)
510081            {
510082              //
510083              // Truncate inode.
510084              //
510085              status = inode_truncate (inode);
510086              if (status != 0)
510087                {
510088                  //
510089                  // Cannot truncate the inode: release it and return.
510090                  // But this error should never happen, because the
510091                  // function 'inode_truncate()' will not return any
510092                  // other value than zero.
510093                  //
510094                  errno_save = errno;
510095                  inode_put (inode);
510096                  errset (errno_save);
510097                  return (-1);
510098                }
```

```
510099            }
510100        }
510101      else
510102        {
510103          //
510104          // The file is to be opened for read, but not for write.
510105          // Try to get inode.
510106          //
510107          inode = path_inode (pid, path);
510108          if (inode == NULL)
510109            {
510110              //
510111              // Cannot open the file.
510112              //
510113              errset (errno);
510114              return (-1);
510115            }
510116        }
510117      //
510118      // An inode was opened: check type and access permissions.
510119      // All file types are good, even directories, as the type
510120      // DIR is implemented through file descriptors.
510121      //
510122      perm = 0;
510123      if (oflags & O_RDONLY) perm |= 4;
510124      if (oflags & O_WRONLY) perm |= 2;
510125      status = inode_check (inode, S_IFMT, perm, ps->uid);
510126      if (status != 0)
510127        {
510128          //
510129          // The file type is not correct or the user does not have
510130          // permissions.
510131          //
510132          return (-1);
510133        }
510134      //
510135      // Allocate the file, inside the file table.
510136      //
510137      file = file_reference (-1);
510138      if (file == NULL)
510139        {
510140          //
510141          // Cannot allocate the file inside the file table: release the
510142          // inode, update 'errno' and return.
510143          //
510144          inode_put (inode);
510145          errset (ENFILE);        // Too many files open in system.
510146          return (-1);
510147        }
510148      //
510149      // Put some data inside the file item. Only options
510150      // O_RDONLY and O_WRONLY are kept here, because the O_APPEND
510151      // is saved inside the file descriptor table.
510152      //
510153      file->references = 1;
510154      file->oflags     = (oflags & (O_RDONLY | O_WRONLY));
510155      file->inode      = inode;
510156      //
510157      // Allocate the file descriptor: variable 'fdn' will be modified
510158      // by the call to 'fd_reference()'.
510159      //
510160      fdn = -1;
510161      fd = fd_reference (pid, &fdn);
510162      if (fd == NULL)
510163        {
510164          //
510165          // Cannot allocate the file descriptor: remove the item from
510166          // file table.
510167          //
510168          file->references = 0;
510169          file->oflags     = 0;
510170          file->inode      = NULL;
510171          //
510172          // Release the inode.
510173          //
510174          inode_put (inode);
510175          //
510176          // Return an error.
510177          //
510178          errset (EMFILE);        // Too many open files.
510179          return (-1);
510180        }
510181      //
510182      // File descriptor allocated: put some data inside the
510183      // file descriptor item.
510184      //
510185      fd->fl_flags = (oflags & (O_RDONLY | O_WRONLY | O_APPEND));
510186      fd->fd_flags = 0;
510187      fd->file = file;
510188      fd->file->offset = 0;
510189      //
510190      // Check if it is a terminal (currently only consoles), if it is
510191      // opened for read and write, and if it have to be set as the
510192      // controlling terminal. This thing is done here because there is
510193      // not a real device driver.
510194      //
510195      if ((S_ISCHR (inode->mode)) &&
510196          (oflags & O_RDONLY)     &&
510197          (oflags & O_WRONLY))
510198        {
510199          //
```



```
510200          // The inode is a character special file (related to a character
510201          // device), opened for read and write!
510202          //
510203          if ((inode->direct[0] & 0xFF00) == (DEV_CONSOLE_MAJOR << 8))
510204            {
510205              //
510206              // It is a terminal (currently only consoles are possible).
510207              // Get the tty reference.
510208              //
510209              tty = tty_reference ((dev_t) inode->direct[0]);
510210              //
510211              // Verify that the terminal is not already the controlling
510212              // terminal of some process group.
510213              //
510214              if (tty->pgrp == 0)
510215                {
510216                  //
510217                  // The terminal is free: verify if the current process
510218                  // needs a controlling terminal.
510219                  //
510220                  if (ps->device_tty == 0 && ps->pgrp == pid)
510221                    {
510222                      //
510223                      // It is a group leader with no controlling
510224                      // terminal: set the controlling terminal.
510225                      //
510226                      ps->device_tty = inode->direct[0];
510227                      tty->pgrp      = ps->pgrp;
510228                    }
510229                }
510230            }
510231        }
510232      //
510233      // Return the file descriptor.
510234      //
510235      return (fdn);
510236  }
```

## kernel/fs/fd_read.c

«

Si veda la sezione i159.3.9.

```
520001  #include <kernel/proc.h>
520002  #include <kernel/k_libc.h>
520003  #include <errno.h>
520004  #include <fcntl.h>
520005  //---------------------------------------------------------------------
520006  ssize_t
520007  fd_read (pid_t pid, int fdn, void *buffer, size_t count, int *eof)
520008  {
520009      fd_t  *fd;
520010      ssize_t  size_read;
520011      //
520012      // Get file descriptor.
520013      //
520014      fd = fd_reference (pid, &fdn);
520015      if (fd              == NULL   ||
520016          fd->file        == NULL   ||
520017          fd->file->inode == NULL )
520018        {
520019          errset (EBADF);                 // Bad file descriptor.
520020          return ((ssize_t) -1);
520021        }
520022      //
520023      // Check if it is opened for read.
520024      //
520025      if (!(fd->file->oflags & O_RDONLY))
520026        {
520027          //
520028          // The file is not opened for read.
520029          //
520030          errset (EINVAL);                // Invalid argument.
520031          return ((ssize_t) -1);
520032        }
520033      //
520034      // It is not a mistake to read a directory, as 'dirent.h' is
520035      // implemented through file descriptors.
520036      //
520037      //  //
520038      //  // Check if it is a directory.
520039      //  //
520040      //  if (fd->file->inode->mode & S_IFDIR)
520041      //    {
520042      //      errset (EISDIR);                // Is a directory.
520043      //      return ((ssize_t) -1);
520044      //    }
520045      //
520046      // Check the kind of file to be read and read it.
520047      //
520048      if  (S_ISBLK (fd->file->inode->mode)
520049        || S_ISCHR (fd->file->inode->mode))
520050        {
520051          //
520052          // A device is to be read.
520053          //
520054          size_read = dev_io (pid, (dev_t) fd->file->inode->direct[0],
520055                              DEV_READ, fd->file->offset, buffer, count,
520056                              eof);
520057        }
520058      else if (S_ISREG (fd->file->inode->mode))
520059        {
```

```
520060          //
520061          // A regular file is to be read.
520062          //
520063          size_read = inode_file_read (fd->file->inode, fd->file->offset,
520064                                       buffer, count, eof);
520065        }
520066      else if (S_ISDIR (fd->file->inode->mode))
520067        {
520068          //
520069          // A directory, is to be read.
520070          //
520071          size_read = inode_file_read (fd->file->inode, fd->file->offset,
520072                                       buffer, count, eof);
520073        }
520074      else
520075        {
520076          //
520077          // Unsupported file type.
520078          //
520079          errset (E_FILE_TYPE_UNSUPPORTED);       //File type unsupported.
520080          return ((ssize_t) -1);
520081        }
520082      //
520083      // Update the file descriptor internal offset.
520084      //
520085      if (size_read > 0)
520086        {
520087          fd->file->offset += size_read;
520088        }
520089      //
520090      // Just return the size read, even if it is an error. Please note
520091      // that a size of zero might tell that it is the end of file, or
520092      // just that the read should be retried.
520093      //
520094      return (size_read);
520095  }
```

kernel/fs/fd_reference.c

«

Si veda la sezione i159.3.10.

```
530001  #include <kernel/proc.h>
530002  #include <kernel/k_libc.h>
530003  #include <errno.h>
530004  //-------------------------------------------------------------------------
530005  fd_t *
530006  fd_reference (pid_t pid, int *fdn)
530007  {
530008      proc_t *ps;
530009      //
530010      // Get process.
530011      //
530012      ps = proc_reference (pid);
530013      //
530014      // See what to do.
530015      //
530016      if (*fdn < 0)
530017        {
530018          //
530019          // Find the first free slot.
530020          //
530021          for (*fdn = 0; *fdn < OPEN_MAX; (*fdn)++)
530022            {
530023              if (ps->fd[*fdn].file == NULL)
530024                {
530025                  return (&(ps->fd[*fdn]));
530026                }
530027            }
530028          *fdn = -1;
530029          return (NULL);
530030        }
530031      else
530032        {
530033          if (*fdn < OPEN_MAX)
530034            {
530035              //
530036              // Might return even a free file descriptor.
530037              //
530038              return (&(ps->fd[*fdn]));
530039            }
530040          else
530041            {
530042              return (NULL);
530043            }
530044        }
530045  }
```

kernel/fs/fd_stat.c

«

Si veda la sezione i159.3.50.

```
540001  #include <kernel/proc.h>
540002  #include <kernel/k_libc.h>
540003  #include <errno.h>
540004  #include <fcntl.h>
540005  //-------------------------------------------------------------------------
540006  int
540007  fd_stat (pid_t pid, int fdn, struct stat *buffer)
540008  {
540009      proc_t   *ps;
```



```
540010      inode_t  *inode;
540011      //
540012      // Get process.
540013      //
540014      ps = proc_reference (pid);
540015      //
540016      // Verify if the file descriptor is valid.
540017      //
540018      if (ps->fd[fdn].file == NULL)
540019        {
540020          errset (EBADF);                 // Bad file descriptor.
540021          return (-1);
540022        }
540023      //
540024      // Reach the inode.
540025      //
540026      inode = ps->fd[fdn].file->inode;
540027      //
540028      // Inode loaded: update the buffer.
540029      //
540030      buffer->st_dev      = inode->sb->device;
540031      buffer->st_ino      = inode->ino;
540032      buffer->st_mode     = inode->mode;
540033      buffer->st_nlink    = inode->links;
540034      buffer->st_uid      = inode->uid;
540035      buffer->st_gid      = inode->gid;
540036      if (S_ISBLK (buffer->st_mode) || S_ISCHR (buffer->st_mode))
540037        {
540038          buffer->st_rdev = inode->direct[0];
540039        }
540040      else
540041        {
540042          buffer->st_rdev = 0;
540043        }
540044      buffer->st_size     = inode->size;
540045      buffer->st_atime    = inode->time;  // All times are the same for
540046      buffer->st_mtime    = inode->time;  // Minix 1 file system.
540047      buffer->st_ctime    = inode->time;  //
540048      buffer->st_blksize  = inode->sb->blksize;
540049      buffer->st_blocks   = inode->blkcnt;
540050      //
540051      // If the inode is a device special file, the 'st_rdev' value is
540052      // taken from the first direct zone (as of Minix 1 organization).
540053      //
540054      if (S_ISBLK(inode->mode) || S_ISCHR(inode->mode))
540055        {
540056          buffer->st_rdev = inode->direct[0];
540057        }
540058      else
540059        {
540060          buffer->st_rdev = 0;
540061        }
540062      //
540063      // Return.
540064      //
540065      return (0);
540066  }
```

kernel/fs/fd_write.c

«

Si veda la sezione i159.3.12.

```
550001  #include <kernel/proc.h>
550002  #include <kernel/k_libc.h>
550003  #include <errno.h>
550004  #include <fcntl.h>
550005  //-------------------------------------------------------------------------
550006  ssize_t
550007  fd_write (pid_t pid, int fdn, const void *buffer, size_t count)
550008  {
550009      proc_t  *ps;
550010      fd_t    *fd;
550011      ssize_t  size_written;
550012      //
550013      // Get process.
550014      //
550015      ps = proc_reference (pid);
550016      //
550017      // Get file descriptor.
550018      //
550019      fd = fd_reference (pid, &fdn);
550020      if (fd              == NULL   ||
550021          fd->file        == NULL   ||
550022          fd->file->inode == NULL )
550023        {
550024          //
550025          // The file descriptor pointer is not valid.
550026          //
550027          errset (EBADF);                 // Bad file descriptor.
550028          return ((ssize_t) -1);
550029        }
550030      //
550031      // Check if it is opened for write.
550032      //
550033      if (!(fd->file->oflags & O_WRONLY))
550034        {
550035          //
550036          // The file is not opened for write.
550037          //
550038          errset (EINVAL);                // Invalid argument.
550039          return ((ssize_t) -1);
```

```
550040        }
550041      //
550042      // Check if it is a directory.
550043      //
550044      if (fd->file->inode->mode & S_IFDIR)
550045        {
550046          errset (EISDIR);                // Is a directory.
550047          return ((ssize_t) -1);
550048        }
550049      //
550050      // It should be a valid type of file to be written. Check if it is
550051      // opened in append mode: if so, must move the write offset to the
550052      // end.
550053      //
550054      if (fd->fl_flags & O_APPEND)
550055        {
550056          fd->file->offset = fd->file->inode->size;
550057        }
550058      //
550059      // Check the kind of file to be written and write it.
550060      //
550061      if (fd->file->inode->mode & S_IFBLK ||
550062          fd->file->inode->mode & S_IFCHR)
550063        {
550064          //
550065          // A device is to be written.
550066          //
550067          size_written = dev_io (pid, (dev_t) fd->file->inode->direct[0],
550068                                 DEV_WRITE, fd->file->offset, buffer,
550069                                 count, NULL);
550070        }
550071      else if (fd->file->inode->mode & S_IFREG)
550072        {
550073          //
550074          // A regular file is to be written.
550075          //
550076          size_written = inode_file_write (fd->file->inode,
550077                                           fd->file->offset,
550078                                           buffer, count);
550079        }
550080      else
550081        {
550082          //
550083          // Unsupported file type.
550084          //
550085          errset (E_FILE_TYPE_UNSUPPORTED);       //File type unsupported.
550086          return ((ssize_t) -1);
550087        }
550088      //
550089      // Update the file descriptor internal offset.
550090      //
550091      if (size_written > 0)
550092        {
550093          fd->file->offset += size_written;
550094        }
550095      //
550096      // Just return the size written, even if it is an error.
550097      //
550098      return (size_written);
550099  }
```

## kernel/fs/file_reference.c

«

Si veda la sezione i159.3.13.

```
560001  #include <kernel/proc.h>
560002  #include <errno.h>
560003  #include <fcntl.h>
560004  //-----------------------------------------------------------------------
560005  file_t *
560006  file_reference (int fno)
560007  {
560008      //
560009      // Check type of request.
560010      //
560011      if (fno < 0)
560012        {
560013          //
560014          // Find a free slot.
560015          //
560016          for (fno = 0; fno < FILE_MAX_SLOTS; fno++)
560017            {
560018              if (file_table[fno].references <= 0)
560019                {
560020                  return (&file_table[fno]);
560021                }
560022            }
560023          return (NULL);
560024        }
560025      else if (fno > FILE_MAX_SLOTS)
560026        {
560027          return (NULL);
560028        }
560029      else
560030        {
560031          return (&file_table[fno]);
560032        }
560033  }
```



## kernel/fs/file_stdio_dev_make.c

«

Si veda la sezione i159.3.14.

```
570001  #include <kernel/proc.h>
570002  #include <errno.h>
570003  #include <fcntl.h>
570004  //-----------------------------------------------------------------------
570005  file_t *
570006  file_stdio_dev_make (dev_t device, mode_t mode, int oflags)
570007  {
570008      inode_t *inode;
570009      file_t  *file;
570010      //
570011      // Try to allocate a device inode.
570012      //
570013      inode = inode_stdio_dev_make (device, mode);
570014      if (inode == NULL)
570015        {
570016          //
570017          // Variable 'errno' is already set by 'inode_stdio_dev_make()'.
570018          //
570019          errset (errno);
570020          return (NULL);
570021        }
570022      //
570023      // Inode allocated: need to allocate the system file item.
570024      //
570025      file = file_reference (-1);
570026      if (file == NULL)
570027        {
570028          //
570029          // Remove the inode and return an error.
570030          //
570031          inode_put (inode);
570032          errset (ENFILE);        // Too many files open in system.
570033          return (NULL);
570034        }
570035      //
570036      // Fill with data the system file item.
570037      //
570038      file->references = 1;
570039      file->oflags     = (oflags & (O_RDONLY | O_WRONLY));
570040      file->inode      = inode;
570041      //
570042      // Return system file pointer.
570043      //
570044      return (file);
570045  }
```

## kernel/fs/file_table.c

«

Si veda la sezione i159.3.13.

```
580001  #include <kernel/fs.h>
580002  //-----------------------------------------------------------------------
580003  file_t file_table[FILE_MAX_SLOTS];
580004  //-----------------------------------------------------------------------
```

## kernel/fs/inode_alloc.c

«

Si veda la sezione i159.3.15.

```
590001  #include <kernel/fs.h>
590002  #include <errno.h>
590003  #include <kernel/k_libc.h>
590004  //-----------------------------------------------------------------------
590005  inode_t *
590006  inode_alloc (dev_t device, mode_t mode, uid_t uid)
590007  {
590008      sb_t        *sb;
590009      inode_t     *inode;
590010      int          m;                     // Index inside the inode map.
590011      int          map_element;
590012      int          map_bit;
590013      int          map_mask;
590014      ino_t        ino;
590015      //
590016      // Check for arguments.
590017      //
590018      if (mode == 0)
590019        {
590020          errset (EINVAL);        // Invalid argument.
590021          return (NULL);
590022        }
590023      //
590024      // Get the super block from the known device.
590025      //
590026      sb = sb_reference (device);
590027      if (sb == NULL)
590028        {
590029          errset (ENODEV);        // No such device.
590030          return (NULL);
590031        }
590032      //
590033      // Find a free inode.
590034      //
590035      while (1)
590036        {
590037          //
```

```
590038          // Scan the inode bit map, to find a free inode
590039          // for new allocation.
590040          //
590041          for (m = 0; m < (SB_MAP_INODE_SIZE * 16); m++)
590042            {
590043              map_element = m / 16;
590044              map_bit     = m % 16;
590045              map_mask    = 1 << map_bit;
590046              if (!(sb->map_inode[map_element] & map_mask))
590047                {
590048                  //
590049                  // Found a free element: change the map to
590050                  // allocate the inode.
590051                  //
590052                  sb->map_inode[map_element] |= map_mask;
590053                  sb->changed = 1;
590054                  ino = m;  // Found a free inode:
590055                  break;          // exit the scan loop.
590056                }
590057            }
590058          //
590059          // Check if the scan was successful.
590060          //
590061          if (ino == 0)
590062            {
590063              errset (ENOSPC);    // No space left on device.
590064              return (NULL);
590065            }
590066          //
590067          // The inode was allocated inside the map in memory.
590068          //
590069          inode = inode_get (device, ino);
590070          if (inode == NULL)
590071            {
590072              errset (ENFILE);    // Too many files open in system.
590073              return (NULL);
590074            }
590075          //
590076          // Verify if the inode is really free: if it isn't, must save
590077          // it to disk.
590078          //
590079          if (inode->size > 0 || inode->links > 0)
590080            {
590081              //
590082              // Strange: should not have a size! Check if there are even
590083              // links. Please note that 255 links (that is -1) is to be
590084              // considered a free inode, marked in a special way for some
590085              // unknown reason. Currently, 'LINK_MAX' is equal to 254,
590086              // for that reason.
590087              //
590088              if (inode->links > 0 && inode->links < LINK_MAX)
590089                {
590090                  //
590091                  // Tell something.
590092                  //
590093                  k_printf ("kernel alert: device %04x: "
590094                            "found \"free\" inode %i "
590095                            "that still has size %i "
590096                            "and %i links!\n",
590097                            device, ino, inode->size, inode->links);
590098                  //
590099                  // The inode must be set again to free, inside
590100                  // the bit map.
590101                  //
590102                  map_element = ino / 16;
590103                  map_bit     = ino % 16;
590104                  map_mask    = 1 << map_bit;
590105                  sb->map_inode[map_element] &= ~map_mask;
590106                  sb->changed = 1;
590107                  //
590108                  // Try to fix: reset all to zero.
590109                  //
590110                  inode->mode      = 0;
590111                  inode->uid       = 0;
590112                  inode->gid       = 0;
590113                  inode->time      = 0;
590114                  inode->links     = 0;
590115                  inode->size      = 0;
590116                  inode->direct[0] = 0;
590117                  inode->direct[1] = 0;
590118                  inode->direct[2] = 0;
590119                  inode->direct[3] = 0;
590120                  inode->direct[4] = 0;
590121                  inode->direct[5] = 0;
590122                  inode->direct[6] = 0;
590123                  inode->indirect1 = 0;
590124                  inode->indirect2 = 0;
590125                  inode->changed   = 1;
590126                  //
590127                  // Save fixed inode to disk.
590128                  //
590129                  inode_put (inode);
590130                  continue;
590131                }
590132              else
590133                {
590134                  //
590135                  // Truncate the inode, save and break.
590136                  //
590137                  inode_truncate (inode);
590138                  inode_save (inode);
```



```
590139                  break;
590140                }
590141            }
590142          else
590143            {
590144              //
590145              // Considering free the inode found.
590146              //
590147              break;
590148            }
590149        }
590150      //
590151      // Put data inside the inode.
590152      //
590153      inode->mode    = mode;
590154      inode->uid     = uid;
590155      inode->gid     = 0;
590156      inode->size    = 0;
590157      inode->time    = k_time (NULL);
590158      inode->links   = 0;
590159      inode->changed = 1;
590160      //
590161      // Save the inode.
590162      //
590163      inode_save (inode);
590164      //
590165      // Return the inode pointer.
590166      //
590167      return (inode);
590168  }
```

## kernel/fs/inode_check.c

Si veda la sezione i159.3.16.

```
600001  #include <kernel/fs.h>
600002  #include <errno.h>
600003  #include <kernel/k_libc.h>
600004  //---------------------------------------------------------------------
600005  int
600006  inode_check (inode_t *inode, mode_t type, int perm, uid_t uid)
600007  {
600008      //
600009      // Ensure that the variable 'type' has only the requested file type.
600010      //
600011      type = (type & S_IFMT);
600012      //
600013      // Check inode argument.
600014      //
600015      if (inode == NULL)
600016        {
600017          errset (EINVAL);        // Invalid argument.
600018          return (-1);
600019        }
600020      //
600021      // The inode is not NULL: verify that the inode is of a type
600022      // allowed (the parameter 'type' can hold more than one
600023      // possibility).
600024      //
600025      if (!(inode->mode & type))
600026        {
600027          errset (E_FILE_TYPE);   // The file type is not
600028          return (-1);            // the expected one.
600029        }
600030      //
600031      // The file type is correct.
600032      //
600033      if (inode->uid != 0 && uid == 0)
600034        {
600035          return (0);             // The root user has all permissions.
600036        }
600037      //
600038      // The user is not root or the inode is owned by root.
600039      //
600040      if (inode->uid == uid)
600041        {
600042          //
600043          // The user own the inode and must check user permissions.
600044          //
600045          perm = (perm << 6);
600046          if ((inode->mode & perm) ^ perm)
600047            {
600048              errset (EACCES);            // Permission denied.
600049              return (-1);
600050            }
600051          else
600052            {
600053              return (0);
600054            }
600055        }
600056      //
600057      // The user does not own the inode: the other permissions are
600058      // checked.
600059      //
600060      if ((inode->mode & perm) ^ perm)
600061        {
600062          errset (EACCES);                // Permission denied.
600063          return (-1);
600064        }
600065      else
600066        {
```

```
600067          return (0);
600068        }
600069  }
```

## kernel/fs/inode_dir_empty.c

«

Si veda la sezione i159.3.17.

```
610001  #include <kernel/fs.h>
610002  #include <errno.h>
610003  #include <kernel/k_libc.h>
610004  //---------------------------------------------------------------------
610005  int
610006  inode_dir_empty (inode_t *inode)
610007  {
610008      off_t        start;
610009      char         buffer[SB_MAX_ZONE_SIZE];
610010      directory_t *dir;
610011      ssize_t      size_read;
610012      int          d;                     // Directory buffer index.
610013      //
610014      // Check argument: must be a directory.
610015      //
610016      if (inode == NULL || !S_ISDIR (inode->mode))
610017        {
610018          errset (EINVAL);                // Invalid argument.
610019          return (0);                     // false
610020        }
610021      //
610022      // Read the directory content: if an item is present (except '.' and
610023      // '..'), the directory is not empty.
610024      //
610025      for (start  = 0;
610026           start <  inode->size;
610027           start += inode->sb->blksize)
610028        {
610029          size_read = inode_file_read (inode, start, buffer,
610030                                       inode->sb->blksize,
610031                                       NULL);
610032          if (size_read < sizeof (directory_t))
610033            {
610034              break;
610035            }
610036          //
610037          // Scan the directory portion just read.
610038          //
610039          dir = (directory_t *) buffer;
610040          //
610041          for (d = 0; d < size_read; d += (sizeof (directory_t)), dir++)
610042            {
610043              if (dir->ino != 0                                   &&
610044                  strncmp (dir->name, ".", NAME_MAX) != 0    &&
610045                  strncmp (dir->name, "..", NAME_MAX) != 0)
610046                {
610047                  //
610048                  // There is an item and the directory is not empty.
610049                  //
610050                  return (0);     // false
610051                }
610052            }
610053        }
610054      //
610055      // Nothing was found; good!
610056      //
610057      return (1);         // true
610058  }
```

## kernel/fs/inode_file_read.c

«

Si veda la sezione i159.3.18.

```
620001  #include <kernel/fs.h>
620002  #include <errno.h>
620003  #include <kernel/k_libc.h>
620004  //---------------------------------------------------------------------
620005  ssize_t
620006  inode_file_read (inode_t *inode, off_t offset,
620007                   void *buffer, size_t count, int *eof)
620008  {
620009      unsigned char *destination = (unsigned char *) buffer;
620010      unsigned char  zone_buffer[SB_MAX_ZONE_SIZE];
620011      blkcnt_t       blkcnt_read;
620012      off_t          off_fzone;   // File zone offset.
620013      off_t          off_buffer;  // Destination buffer offset.
620014      ssize_t        size_read;   // Byte transfer counter.
620015      zno_t          fzone;
620016      off_t          off_end;
620017      //
620018      // The inode pointer must be valid, and
620019      // the start byte must be positive.
620020      //
620021      if (inode == NULL || offset < 0)
620022        {
620023          errset (EINVAL);        // Invalid argument.
620024          return ((ssize_t) -1);
620025        }
620026      //
620027      // Check if the start address is inside the file size. This is not
620028      // an error, but zero bytes are read and '*eof' is set. Otherwise,
620029      // '*eof' is reset.
620030      //
620031      if (offset >= inode->size)
620032        {
620033          (eof != NULL)? *eof = 1: 0;
620034          return (0);
620035        }
620036      else
620037        {
620038          (eof != NULL)? *eof = 0: 0;
620039        }
620040      //
620041      // Adjust, if necessary, the size of read, because it cannot be
620042      // larger than the actual file size. The variable 'off_end' is
620043      // used to calculate the position *after* the requested read.
620044      // Remember that the first file position is byte zero; so,
620045      // the byte index inside the file goes from zero to inode->size -1.
620046      //
620047      off_end  = offset;
620048      off_end += count;
620049      if (off_end > inode->size)
620050        {
620051          count = (inode->size - offset);
620052        }
620053      //
620054      // Read the first file-zone inside the zone buffer.
620055      //
620056      fzone       = offset / inode->sb->blksize;
620057      off_fzone   = offset % inode->sb->blksize;
620058      blkcnt_read = inode_fzones_read (inode, fzone, zone_buffer,
620059                                       (blkcnt_t) 1);
620060      if (blkcnt_read <= 0)
620061        {
620062          //
620063          // Sorry!
620064          //
620065          return (0);                 // Zero bytes read!
620066        }
620067      //
620068      // The first file-zone was read: copy it inside the destination
620069      // buffer and continue reading the other zones needed. Variables
620070      // 'off_buffer' (destination buffer index) and 'size_read' (copy
620071      // byte counter) must be reset here. Variable 'off_fzone' is already
620072      // set with the initial offset inside 'zone_buffer'.
620073      //
620074      off_buffer = 0;
620075      size_read  = 0;
620076      //
620077      while (count)
620078        {
620079          //
620080          // Copy the zone buffer into the destination. Variables
620081          // 'off_fzone', 'off_buffer' and 'size_read' must not be
620082          // initialized inside the loop.
620083          //
620084          for (; off_fzone < inode->sb->blksize && count > 0;
620085               off_fzone++, off_buffer++, size_read++,
620086               count--, offset++)
620087            {
620088              destination[off_buffer] = zone_buffer[off_fzone];
620089            }
620090          //
620091          // If not all the bytes are copied, read the next file-zone.
620092          //
620093          if (count)
620094            {
620095              //
620096              // Read another file-zone inside the zone buffer.
620097              // Again, the function 'inode_fzones_read()' might
620098              // return a null pointer, but the variable 'errno' tells if
620099              // it is really an error. For this reason, the variable
620100              // 'errno' must be reset before the read, and checked after
620101              // it.
620102              //
620103              fzone       = offset / inode->sb->blksize;
620104              off_fzone   = offset % inode->sb->blksize;
620105              blkcnt_read = inode_fzones_read (inode, fzone, zone_buffer,
620106                                               (blkcnt_t) 1);
620107              if (blkcnt_read <= 0)
620108                {
620109                  //
620110                  // Sorry: only 'size_read' bytes read!
620111                  //
620112                  return (size_read);
620113                }
620114            }
620115        }
620116      //
620117      // The requested size was read completely.
620118      //
620119      return (size_read);
620120  }
```

## kernel/fs/inode_file_write.c

«

Si veda la sezione i159.3.19.

```
630001  #include <kernel/fs.h>
630002  #include <errno.h>
630003  #include <kernel/k_libc.h>
630004  //---------------------------------------------------------------------
630005  ssize_t
```

```
630006 inode_file_write (inode_t *inode, off_t offset, void *buffer,
630007                   size_t count)
630008 {
630009     unsigned char *buffer_source = (unsigned char *) buffer;
630010     unsigned char  buffer_zone[SB_MAX_ZONE_SIZE];
630011     off_t          off_fzone;    // File zone offset.
630012     off_t          off_source;   // Source buffer offset.
630013     ssize_t        size_copied;  // Byte transfer counter.
630014     ssize_t        size_written; // Byte written counter.
630015     zno_t          fzone;
630016     zno_t          zone;
630017     blkcnt_t       blkcnt_read;
630018     int            status;
630019     //
630020     // The inode pointer must be valid, and
630021     // the start byte must be positive.
630022     //
630023     if (inode == NULL || offset < 0)
630024       {
630025         errset (EINVAL);        // Invalid argument.
630026         return ((ssize_t) -1);
630027       }
630028     //
630029     // Read a zone, modify it with the source buffer, then write it back
630030     // and continue reading and writing other zones if needed.
630031     //
630032     for (size_written = 0, off_source = 0, size_copied = 0;
630033          count > 0; size_written += size_copied)
630034       {
630035         //
630036         // Read the next file-zone inside the zone buffer: the function
630037         // 'inode_zone()' is used to create automatically the zone, if
630038         // it does not exist.
630039         //
630040         fzone     = offset / inode->sb->blksize;
630041         off_fzone = offset % inode->sb->blksize;
630042         zone      = inode_zone (inode, fzone, 1);
630043         if (zone == 0)
630044           {
630045             //
630046             // Return previously written bytes. The variable 'errno' is
630047             // already set by 'inode_zone()'.
630048             //
630049             return (size_written);
630050           }
630051         blkcnt_read = inode_fzones_read (inode, fzone, buffer_zone,
630052                                          (blkcnt_t) 1);
630053         if (blkcnt_read <= 0)
630054           {
630055             //
630056             // Even if the value is zero, there is a problem reading the
630057             // zone to be overwritten (because 'inode_zone()' should
630058             // have already created such zone). The variable 'errno' is
630059             // already set by 'inode_fzones_read()'.
630060             //
630061             return ((ssize_t) -1);
630062           }
630063         //
630064         // The zone was successfully loaded inside the buffer: overwrite
630065         // the zone buffer with the source buffer.
630066         //
630067         for (size_copied = 0;
630068              off_fzone < inode->sb->blksize && count > 0;
630069              off_fzone++, off_source++, size_copied++, count--,
630070              offset++)
630071           {
630072             buffer_zone[off_fzone] = buffer_source[off_source];
630073           }
630074         //
630075         // Save the zone.
630076         //
630077         status = zone_write (inode->sb, zone, buffer_zone);
630078         if (status != 0)
630079           {
630080             //
630081             // Cannot save the zone: return the size already written.
630082             // The variable 'errno' is already set by 'zone_write()'.
630083             //
630084             return (size_written);
630085           }
630086         //
630087         // Zone saved: update the file size if necessary (and the inode
630088         // too).
630089         //
630090         if (inode->size <= offset)
630091           {
630092             inode->size    = offset;
630093             inode->changed = 1;
630094             inode_save (inode);
630095           }
630096       }
630097     //
630098     // All done successfully: return the value.
630099     //
630100     return (size_written);
630101 }
```



## kernel/fs/inode_free.c

«

Si veda la sezione i159.3.20.

```
640001 #include <kernel/fs.h>
640002 #include <errno.h>
640003 #include <kernel/k_libc.h>
640004 //-----------------------------------------------------------------------
640005 int
640006 inode_free (inode_t *inode)
640007 {
640008     int      map_element;
640009     int      map_bit;
640010     int      map_mask;
640011     //
640012     if (inode == NULL)
640013       {
640014         errset (EINVAL);        // Invalid argument.
640015         return (-1);
640016       }
640017     //
640018     map_element = inode->ino / 16;
640019     map_bit     = inode->ino % 16;
640020     map_mask    = 1 << map_bit;
640021     //
640022     if (inode->sb->map_inode[map_element] & map_mask)
640023       {
640024         inode->sb->map_inode[map_element] -= map_mask;
640025         inode->sb->changed = 1;
640026       }
640027     //
640028     inode->mode       = 0;
640029     inode->uid        = 0;
640030     inode->gid        = 0;
640031     inode->size       = 0;
640032     inode->time       = 0;
640033     inode->links      = 0;
640034     inode->changed    = 1;
640035     inode->references = 0;
640036     //
640037     return (inode_save (inode));
640038 }
```

## kernel/fs/inode_fzones_read.c

«

Si veda la sezione i159.3.21.

```
650001 #include <kernel/fs.h>
650002 #include <errno.h>
650003 #include <kernel/k_libc.h>
650004 //-----------------------------------------------------------------------
650005 blkcnt_t
650006 inode_fzones_read (inode_t *inode, zno_t zone_start,
650007                    void *buffer, blkcnt_t blkcnt)
650008 {
650009     unsigned char *destination = (unsigned char *) buffer;
650010     int            status;        // 'zone_read()' return value.
650011     blkcnt_t       blkcnt_read;   // Zone counter/index.
650012     zno_t          zone;
650013     zno_t          fzone;
650014     //
650015     // Read the zones into the destination buffer.
650016     //
650017     for (blkcnt_read = 0, fzone = zone_start;
650018          blkcnt_read < blkcnt;
650019          blkcnt_read++, fzone++)
650020       {
650021         //
650022         // Calculate the zone number, from the file-zone, reading the
650023         // inode. If a zone is not really allocated, the result is zero
650024         // and is valid.
650025         //
650026         zone = inode_zone (inode, fzone, 0);
650027         if (zone == ((zno_t) -1))
650028           {
650029             //
650030             // This is an error. Return the read zones quantity.
650031             //
650032             return (blkcnt_read);
650033           }
650034         //
650035         // Update the destination buffer pointer.
650036         //
650037         destination += (blkcnt_read * inode->sb->blksize);
650038         //
650039         // Read the zone inside the destination buffer, but if the zone
650040         // is zero, a zeroed zone must be filled.
650041         //
650042         if (zone == 0)
650043           {
650044             memset (destination, 0, (size_t) inode->sb->blksize);
650045           }
650046         else
650047           {
650048             status = zone_read (inode->sb, zone, destination);
650049             if (status != 0)
650050               {
650051                 //
650052                 // Could not read the requested zone: return the zones
650053                 // read correctly.
650054                 //
```

```
650055                  errset (EIO);               // I/O error.
650056                  return (blkcnt_read);
650057                }
650058            }
650059        }
650060      //
650061      // All zones read correctly inside the buffer.
650062      //
650063      return (blkcnt_read);
650064  }
```

## kernel/fs/inode_fzones_write.c

«

Si veda la sezione i159.3.21.

```
660001  #include <kernel/fs.h>
660002  #include <errno.h>
660003  #include <kernel/k_libc.h>
660004  //-----------------------------------------------------------------------
660005  blkcnt_t
660006  inode_fzones_write (inode_t *inode, zno_t zone_start, void *buffer,
660007                      blkcnt_t blkcnt)
660008  {
660009      unsigned char *source = (unsigned char *) buffer;
660010      int            status;              // 'zone_read()' return value.
660011      blkcnt_t       blkcnt_written;      // Written zones counter.
660012      zno_t          zone;
660013      zno_t          fzone;
660014      //
660015      // Write the zones into the destination buffer.
660016      //
660017      for (blkcnt_written = 0, fzone = zone_start;
660018           blkcnt_written < blkcnt;
660019           blkcnt_written++, fzone++)
660020        {
660021          //
660022          // Find real zone from file-zone.
660023          //
660024          zone = inode_zone (inode, fzone, 1);
660025          if (zone == 0 || zone == ((zno_t) -1))
660026            {
660027              //
660028              // Function 'inode_zone()' should allocate automatically
660029              // a missing zone and should return a valid zone or
660030              // (zno_t) -1. Anyway, even if a zero zone is returned,
660031              // it is an error. Return the 'blkcnt_written' value.
660032              //
660033              return (blkcnt_written);
660034            }
660035          //
660036          // Update the source buffer pointer for the next zone write.
660037          //
660038          source += (blkcnt_written * inode->sb->blksize);
660039          //
660040          // Write the zone from the buffer content.
660041          //
660042          status  = zone_write (inode->sb, zone, source);
660043          if (status != 0)
660044            {
660045              //
660046              // Cannot write the zone. Return 'size_written_zone' value.
660047              //
660048              return (blkcnt_written);
660049            }
660050        }
660051      //
660052      // All zones read correctly inside the buffer.
660053      //
660054      return (blkcnt_written);
660055  }
```

## kernel/fs/inode_get.c

«

Si veda la sezione i159.3.23.

```
670001  #include <kernel/fs.h>
670002  #include <errno.h>
670003  #include <kernel/k_libc.h>
670004  #include <kernel/devices.h>
670005  //-----------------------------------------------------------------------
670006  inode_t *
670007  inode_get (dev_t device, ino_t ino)
670008  {
670009      sb_t             *sb;
670010      inode_t          *inode;
670011      unsigned long int start;
670012      size_t            size;
670013      ssize_t           n;
670014      int               status;
670015      //
670016      // Verify if the root file system inode was requested.
670017      //
670018      if (device == 0 && ino == 1)
670019        {
670020          //
670021          // Get root file system inode.
670022          //
670023          inode = inode_reference (device, ino);
670024          if (inode == NULL)
670025            {
```



```
670026              //
670027              // The file system root directory inode is not yet loaded:
670028              // get the first super block.
670029              //
670030              sb = sb_reference ((dev_t) 0);
670031              if (sb == NULL || sb->device == 0)
670032                {
670033                  //
670034                  // This error should never happen.
670035                  //
670036                  errset (EUNKNOWN);      // Unknown error.
670037                  return (NULL);
670038                }
670039              //
670040              // Load the file system root directory inode (recursive
670041              // call).
670042              //
670043              inode = inode_get (sb->device, (ino_t) 1);
670044              if (inode == NULL)
670045                {
670046                  //
670047                  // This error should never happen.
670048                  //
670049                  return (NULL);
670050                }
670051              //
670052              // Return the directory inode.
670053              //
670054              return (inode);
670055            }
670056          else
670057            {
670058              //
670059              // The file system root directory inode is already
670060              // available.
670061              //
670062              if (inode->references >= INODE_MAX_REFERENCES)
670063                {
670064                  errset (ENFILE); // Too many files open in system.
670065                  return (NULL);
670066                }
670067              else
670068                {
670069                  inode->references++;
670070                  return (inode);
670071                }
670072            }
670073        }
670074      //
670075      // A common device-inode pair was requested: try to find an already
670076      // cached inode.
670077      //
670078      inode = inode_reference (device, ino);
670079      if (inode != NULL)
670080        {
670081          if (inode->references >= INODE_MAX_REFERENCES)
670082            {
670083              errset (ENFILE);    // Too many files open in system.
670084              return (NULL);
670085            }
670086          else
670087            {
670088              inode->references++;
670089              return (inode);
670090            }
670091        }
670092      //
670093      // The inode is not yet available: get super block.
670094      //
670095      sb = sb_reference (device);
670096      if (sb == NULL)
670097        {
670098          errset (ENODEV);        // No such device.
670099          return (NULL);
670100        }
670101      //
670102      // The super block is available, but the inode is not yet cached.
670103      // Verify if the inode map reports it as allocated.
670104      //
670105      status = sb_inode_status (sb, ino);
670106      if (!status)
670107        {
670108          //
670109          // The inode is not allocated and cannot be loaded.
670110          //
670111          errset (ENOENT);        // No such file or directory.
670112          return (NULL);
670113        }
670114      //
670115      // The inode was not already cached, but is considered as allocated
670116      // inside the inode map. Find a free slot to load the inode inside
670117      // the inode table (in memory).
670118      //
670119      inode = inode_reference ((dev_t) -1, (ino_t) -1);
670120      if (inode == NULL)
670121        {
670122          errset (ENFILE);        // Too many files open in system.
670123          return (NULL);
670124        }
670125      //
670126      // A free inode slot was found. The inode must be loaded.
```

```
670127      // Calculate the memory inode size, to be saved inside the file
670128      // system: the administrative inode data, as it is saved inside
670129      // the file system. The 'inode_t' type is bigger than the real
670130      // inode administrative size, because it contains more data, that is
670131      // not saved on disk.
670132      //
670133      size   = offsetof (inode_t, sb);
670134      //
670135      // Calculating start position for read.
670136      //
670137      // [1] Boot block.
670138      // [2] Super block.
670139      // [3] Inode bit map.
670140      // [4] Zone bit map.
670141      // [5] Previous inodes: consider that the inode zero is
670142      //     present in the inode map, but not in the inode
670143      //     table.
670144      //
670145      start  = 1024;                          // [1]
670146      start += 1024;                          // [2]
670147      start += (sb->map_inode_blocks * 1024);  // [3]
670148      start += (sb->map_zone_blocks * 1024);   // [4]
670149      start += ((ino -1) * size);              // [5]
670150      //
670151      // Read inode from disk.
670152      //
670153      n = dev_io ((pid_t) -1, device, DEV_READ, start, inode, size, NULL);
670154      if (n != size)
670155        {
670156          errset (EIO);                   // I/O error.
670157          return (NULL);
670158        }
670159      //
670160      // The inode was read: add some data to the working copy in memory.
670161      //
670162      inode->sb            = sb;
670163      inode->sb_attached   = NULL;
670164      inode->ino           = ino;
670165      inode->references    = 1;
670166      inode->changed       = 0;
670167      //
670168      inode->blkcnt        = inode->size;
670169      inode->blkcnt       /= sb->blksize;
670170      if (inode->size % sb->blksize)
670171        {
670172          inode->blkcnt++;
670173        }
670174      //
670175      // Return the inode pointer.
670176      //
670177      return (inode);
670178  }
```

## kernel/fs/inode_put.c

«

Si veda la sezione i159.3.24.

```
680001  #include <kernel/fs.h>
680002  #include <errno.h>
680003  #include <kernel/k_libc.h>
680004  //------------------------------------------------------------------------
680005  int
680006  inode_put (inode_t *inode)
680007  {
680008      int        status;
680009      //
680010      // Check for valid argument.
680011      //
680012      if (inode == NULL)
680013        {
680014          errset (EINVAL);                // Invalid argument.
680015          return (-1);
680016        }
680017      //
680018      // Check for valid references.
680019      //
680020      if (inode->references <= 0)
680021        {
680022          errset (EUNKNOWN);              // Cannot put an inode with
680023          return (-1);                    // zero or negative references.
680024        }
680025      //
680026      // Debug.
680027      //
680028      if (inode->sb->device == 0 && inode->ino != 0)
680029        {
680030          k_printf ("kernel alert: trying to close inode with device "
680031                    "zero, but a number different than zero!\n");
680032          errset (EUNKNOWN);              // Cannot put an inode with
680033          return (-1);                    // zero or negative references.
680034        }
680035      //
680036      // There is at least one reference: now the references value is
680037      // reduced.
680038      //
680039      inode->references--;
680040      inode->changed = 1;
680041      //
680042      // If 'inode->ino' is zero, it means that the inode was created in
680043      // memory, but there is no file system for it. For example, it might
680044      // be a standard I/O inode create automatically for a process.
```



```
680045      // Inodes with number zero cannot be removed from a file system.
680046      //
680047      if (inode->ino == 0)
680048        {
680049          //
680050          // Nothing to do: just return.
680051          //
680052          return (0);
680053        }
680054      //
680055      //  References counter might be zero.
680056      //
680057      if (inode->references == 0)
680058        {
680059          //
680060          // Check if the inode is to be deleted (until there are
680061          // run time references, the inode cannot be removed).
680062          //
680063          if (inode->links == 0
680064           || (S_ISDIR (inode->mode) && inode->links == 1))
680065            {
680066              //
680067              // The inode has no more run time references and file system
680068              // links are also zero (or one for a directory): remove it!
680069              //
680070              status = inode_truncate (inode);
680071              if (status != 0)
680072                {
680073                  k_perror (NULL);
680074                }
680075              //
680076              inode_free (inode);
680077              return (0);
680078            }
680079        }
680080      //
680081      // Save inode to disk and return.
680082      //
680083      return (inode_save (inode));
680084  }
```

## kernel/fs/inode_reference.c

«

Si veda la sezione i159.3.25.

```
690001  #include <kernel/fs.h>
690002  #include <errno.h>
690003  #include <kernel/k_libc.h>
690004  //------------------------------------------------------------------------
690005  inode_t *
690006  inode_reference (dev_t device, ino_t ino)
690007  {
690008      int   s;                            // Slot index.
690009      sb_t *sb_table = sb_reference (0);
690010      //
690011      // If device is zero, and inode is zero, a reference to the whole
690012      // table is returned.
690013      //
690014      if (device == 0 && ino == 0)
690015        {
690016          return (inode_table);
690017        }
690018      //
690019      // If device is ((dev_t) -1) and the inode is ((ino_t) -1), a
690020      // reference to a free inode slot is returned.
690021      //
690022      if (device == (dev_t) -1 && ino == ((ino_t) -1))
690023        {
690024          for (s = 0; s < INODE_MAX_SLOTS; s++)
690025            {
690026              if (inode_table[s].references == 0)
690027                {
690028                  return (&inode_table[s]);
690029                }
690030            }
690031          return (NULL);
690032        }
690033      //
690034      // If device is zero and the inode is 1, a reference to the root
690035      // directory inode is returned.
690036      //
690037      if (device == 0 && ino == 1)
690038        {
690039          //
690040          // The super block table is to be scanned.
690041          //
690042          for (device = 0, s = 0; s < SB_MAX_SLOTS; s++)
690043            {
690044              if (sb_table[s].device != 0                &&
690045                  sb_table[s].inode_mounted_on == NULL)
690046                {
690047                  device = sb_table[s].device;
690048                  break;
690049                }
690050            }
690051          if (device == 0)
690052            {
690053              errset (E_CANNOT_FIND_ROOT_DEVICE);
690054              return (NULL);
690055            }
690056          //
```

```
690057          // Scan the inode table to find inode 1 and the same device.
690058          //
690059          for (s = 0; s < INODE_MAX_SLOTS; s++)
690060            {
690061              if (inode_table[s].sb->device == device     &&
690062                  inode_table[s].ino == 1)
690063                {
690064                  return (&inode_table[s]);
690065                }
690066            }
690067          //
690068          // Cannot find a root file system inode.
690069          //
690070          errset (E_CANNOT_FIND_ROOT_INODE);
690071          return (NULL);
690072        }
690073      //
690074      // A device and an inode number were selected: find the inode
690075      // associated to it.
690076      //
690077      for (s = 0; s < INODE_MAX_SLOTS; s++)
690078        {
690079          if (inode_table[s].sb->device == device &&
690080              inode_table[s].ino == ino)
690081            {
690082              return (&inode_table[s]);
690083            }
690084        }
690085      //
690086      // The inode was not found.
690087      //
690088      return (NULL);
690089  }
```

kernel/fs/inode_save.c

«

Si veda la sezione i159.3.26.

```
700001  #include <kernel/fs.h>
700002  #include <errno.h>
700003  #include <kernel/k_libc.h>
700004  #include <kernel/devices.h>
700005  //-------------------------------------------------------------------------
700006  int
700007  inode_save (inode_t *inode)
700008  {
700009      size_t            size;
700010      unsigned long int start;
700011      ssize_t           n;
700012      //
700013      // Check for valid argument.
700014      //
700015      if (inode == NULL)
700016        {
700017          errset (EINVAL);               // Invalid argument.
700018          return (-1);
700019        }
700020      //
700021      // If the inode number is zero, no file system is involved!
700022      //
700023      if (inode->ino == 0)
700024        {
700025          return (0);
700026        }
700027      //
700028      // Save the super block to disk.
700029      //
700030      sb_save (inode->sb);
700031      //
700032      // Save the inode to disk.
700033      //
700034      if (inode->changed)
700035        {
700036          size   = offsetof (inode_t, sb);
700037          //
700038          // Calculating start position for write.
700039          //
700040          // [1] Boot block.
700041          // [2] Super block.
700042          // [3] Inode bit map.
700043          // [4] Zone bit map.
700044          // [5] Previous inodes: consider that the inode zero is
700045          //     present in the inode map, but not in the inode
700046          //     table.
700047          //
700048          start  = 1024;                                  // [1]
700049          start += 1024;                                  // [2]
700050          start += (inode->sb->map_inode_blocks * 1024);  // [3]
700051          start += (inode->sb->map_zone_blocks * 1024);   // [4]
700052          start += ((inode->ino -1) * size);              // [5]
700053          //
700054          // Write the inode.
700055          //
700056          n = dev_io ((pid_t) -1, inode->sb->device, DEV_WRITE, start,
700057                      inode, size, NULL);
700058          //
700059          inode->changed = 0;
700060        }
700061      return (0);
700062  }
```



kernel/fs/inode_stdio_dev_make.c

«

Si veda la sezione i159.3.27.

```
710001  #include <kernel/fs.h>
710002  #include <errno.h>
710003  #include <kernel/k_libc.h>
710004  //-------------------------------------------------------------------------
710005  inode_t *
710006  inode_stdio_dev_make (dev_t device, mode_t mode)
710007  {
710008      inode_t     *inode;
710009      //
710010      // Check for arguments.
710011      //
710012      if (mode == 0 || device == 0)
710013        {
710014          errset (EINVAL);        // Invalid argument.
710015          return (NULL);
710016        }
710017      //
710018      // Find a free inode.
710019      //
710020      inode = inode_reference ((dev_t) -1, (ino_t) -1);
710021      if (inode == NULL)
710022        {
710023          //
710024          // No free slot available.
710025          //
710026          errset (ENFILE);        // Too many files open in system.
710027          return (NULL);
710028        }
710029      //
710030      // Put data inside the inode. Please note that 'inode->ino' must be
710031      // zero, because it is necessary to recognize it as an internal
710032      // inode with no file system. Otherwise, with a value different than
710033      // zero, 'inode_put()' will try to remove it. [*]
710034      //
710035      inode->mode          = mode;
710036      inode->uid           = 0;
710037      inode->gid           = 0;
710038      inode->size          = 0;
710039      inode->time          = k_time (NULL);
710040      inode->links         = 0;
710041      inode->direct[0]     = device;
710042      inode->direct[1]     = 0;
710043      inode->direct[2]     = 0;
710044      inode->direct[3]     = 0;
710045      inode->direct[4]     = 0;
710046      inode->direct[5]     = 0;
710047      inode->direct[6]     = 0;
710048      inode->indirect1     = 0;
710049      inode->indirect2     = 0;
710050      inode->sb_attached   = NULL;
710051      inode->sb            = 0;
710052      inode->ino           = 0;   // Must be zero. [*]
710053      inode->blkcnt        = 0;
710054      inode->references    = 1;
710055      inode->changed       = 0;
710056      //
710057      // Add all access permissions.
710058      //
710059      inode->mode       |= (S_IRWXU|S_IRWXG|S_IRWXO);
710060      //
710061      // Return the inode pointer.
710062      //
710063      return (inode);
710064  }
```

kernel/fs/inode_table.c

«

Si veda la sezione i159.3.25.

```
720001  #include <kernel/fs.h>
720002  //-------------------------------------------------------------------------
720003  inode_t inode_table[INODE_MAX_SLOTS];
```

kernel/fs/inode_truncate.c

«

Si veda la sezione i159.3.28.

```
730001  #include <kernel/fs.h>
730002  #include <errno.h>
730003  #include <kernel/k_libc.h>
730004  //-------------------------------------------------------------------------
730005  int
730006  inode_truncate (inode_t *inode)
730007  {
730008    unsigned int indirect_zones;
730009    zno_t        zone_table1[INODE_MAX_INDIRECT_ZONES];
730010    zno_t        zone_table2[INODE_MAX_INDIRECT_ZONES];
730011    unsigned int i;             // Direct index.
730012    unsigned int i0;            // Single indirect index.
730013    unsigned int i1;            // Double indirect first index.
730014    unsigned int i2;            // Double indirect second index.
730015    int          status;        // 'zone_read()' return value.
730016    //
730017    // Calculate how many indirect zone numbers are stored inside
730018    // a zone: it depends on the zone size.
730019    //
```

```
730020    indirect_zones  = inode->sb->blksize / 2;
730021    //
730022    // Scan and release direct zones. Errors are ignored.
730023    //
730024    for (i = 0; i < 7; i++)
730025      {
730026        zone_free (inode->sb, inode->direct[i]);
730027        inode->direct[i] = 0;
730028      }
730029    //
730030    // Scan single indirect zones, if present.
730031    //
730032    if (inode->blkcnt > 7 && inode->indirect1 != 0)
730033      {
730034        //
730035        // There is a single indirect table to load. Errors are
730036        // almost ignored.
730037        //
730038        status = zone_read (inode->sb, inode->indirect1, zone_table1);
730039        if (status == 0)
730040          {
730041            //
730042            // Scan the table and remove zones.
730043            //
730044            for (i0 = 0; i0 < indirect_zones; i0++)
730045              {
730046                zone_free (inode->sb, zone_table1[i0]);
730047              }
730048          }
730049        //
730050        // Remove indirect table too.
730051        //
730052        zone_free (inode->sb, inode->indirect1);
730053        //
730054        // Clear single indirect reference inside the inode.
730055        //
730056        inode->indirect1 = 0;
730057      }
730058    //
730059    // Scan double indirect zones, if present.
730060    //
730061    if (   inode->blkcnt > (7+indirect_zones)
730062        && inode->indirect2 != 0)
730063      {
730064        //
730065        // There is a double indirect table to load. Errors are
730066        // almost ignored.
730067        //
730068        status = zone_read (inode->sb, inode->indirect2, zone_table1);
730069        if (status == 0)
730070          {
730071            //
730072            // Scan the table and get second level indirection.
730073            //
730074            for (i1 = 0; i1 < indirect_zones; i1++)
730075              {
730076                if ((inode->blkcnt > (7+indirect_zones+indirect_zones*i1))
730077                    && zone_table1[i1] != 0)
730078                  {
730079                    //
730080                    // There is a second level table to load.
730081                    //
730082                    status = zone_read (inode->sb, zone_table1[i1],
730083                                        zone_table2);
730084                    if (status == 0)
730085                      {
730086                        //
730087                        // Release zones.
730088                        //
730089                        for (i2 = 0;
730090                             i2 < indirect_zones &&
730091    (inode->blkcnt > (7+indirect_zones+indirect_zones*i1+i2));
730092                             i2++)
730093                          {
730094                            zone_free (inode->sb, zone_table2[i2]);
730095                          }
730096                        //
730097                        // Remove second level indirect table.
730098                        //
730099                        zone_free (inode->sb, zone_table1[i1]);
730100                      }
730101                  }
730102              }
730103            //
730104            // Remove first level indirect table.
730105            //
730106            zone_free (inode->sb, inode->indirect2);
730107          }
730108        //
730109        // Clear single indirect reference inside the inode.
730110        //
730111        inode->indirect2 = 0;
730112      }
730113    //
730114    // Update super block and inode data.
730115    //
730116    sb_save (inode->sb);
730117    inode->size = 0;
730118    inode->changed = 1;
730119    inode_save (inode);
730120    //
```



```
730121    // Successful return.
730122    //
730123    return (0);
730124  }
```

## kernel/fs/inode_zone.c

«

Si veda la sezione i159.3.29.

```
740001  #include <kernel/fs.h>
740002  #include <errno.h>
740003  #include <kernel/k_libc.h>
740004  //---------------------------------------------------------------------
740005  zno_t
740006  inode_zone (inode_t *inode, zno_t fzone, int write)
740007  {
740008    unsigned int indirect_zones;
740009    unsigned int allocated_zone;
740010    zno_t        zone_table[INODE_MAX_INDIRECT_ZONES];
740011    char         buffer[SB_MAX_ZONE_SIZE];
740012    unsigned int i0;            // Single indirect index.
740013    unsigned int i1;            // Double indirect first index.
740014    unsigned int i2;            // Double indirect second index.
740015    int          status;
740016    zno_t        zone_second;   // Second level table zone.
740017    //
740018    // Calculate how many indirect zone numbers are stored inside
740019    // a zone: it depends on the zone size.
740020    //
740021    indirect_zones  = inode->sb->blksize / 2;
740022    //
740023    // Convert file-zone number into a zone number.
740024    //
740025    if (fzone < 7)
740026      {
740027        //
740028        // 0 <= fzone <= 6
740029        // The zone number is inside the direct zone references.
740030        // Verify to have such zone.
740031        //
740032        if (inode->direct[fzone] == 0)
740033          {
740034            //
740035            // There is not such zone, but we do not consider
740036            // it an error, because a file can be not contiguous.
740037            //
740038            if (!write)
740039              {
740040                return ((zno_t) 0);
740041              }
740042            //
740043            // Must be allocated.
740044            //
740045            allocated_zone = zone_alloc (inode->sb);
740046            if (allocated_zone == 0)
740047              {
740048                //
740049                // Cannot allocate the zone. The variable ‘errno’ is
740050                // set by ‘zone_alloc()’.
740051                //
740052                return ((zno_t) -1);
740053              }
740054            //
740055            // The zone is allocated: clear the zone and save.
740056            //
740057            memset (buffer, 0, SB_MAX_ZONE_SIZE);
740058            status = zone_write (inode->sb, allocated_zone, buffer);
740059            if (status < 0)
740060              {
740061                //
740062                // Cannot overwrite the zone. The variable ‘errno’ is
740063                // set by ‘zone_write()’.
740064                //
740065                return ((zno_t) -1);
740066              }
740067            //
740068            // The zone is allocated and cleared: save the inode.
740069            //
740070            inode->direct[fzone] = allocated_zone;
740071            inode->changed = 1;
740072            status = inode_save (inode);
740073            if (status != 0)
740074              {
740075                //
740076                // Cannot save the inode. The variable ‘errno’ is
740077                // set ‘inode_save()’.
740078                //
740079                return ((zno_t) -1);
740080              }
740081          }
740082        //
740083        // The zone is there: return it.
740084        //
740085        return (inode->direct[fzone]);
740086      }
740087    if (fzone < 7 + indirect_zones)
740088      {
740089        //
740090        // 7 <= fzone <= (6 + indirect_zones)
740091        // The zone number is inside the single indirect zone
740092        // references: verify to have the indirect zone table.
```

```
740093        //
740094        if (inode->indirect1 == 0)
740095          {
740096            //
740097            // There is not such zone, but it is not an error.
740098            //
740099            if (!write)
740100              {
740101                return ((zno_t) 0);
740102              }
740103            //
740104            // The first level of indirection must be initialized.
740105            //
740106            allocated_zone = zone_alloc (inode->sb);
740107            if (allocated_zone == 0)
740108              {
740109                //
740110                // Cannot allocate the zone for the indirection table:
740111                // this is an error and the ‘errno’ value is produced
740112                // by ‘zone_alloc()’.
740113                //
740114                return ((zno_t) -1);
740115              }
740116            //
740117            // The zone for the indirection table is allocated:
740118            // clear the zone and save.
740119            //
740120            memset (buffer, 0, SB_MAX_ZONE_SIZE);
740121            status = zone_write (inode->sb, allocated_zone, buffer);
740122            if (status < 0)
740123              {
740124                //
740125                // Cannot overwrite the zone. The variable ‘errno’ is
740126                // set by ‘zone_write()’.
740127                //
740128                return ((zno_t) -1);
740129              }
740130            //
740131            // The indirection table zone is allocated and cleared:
740132            // save the inode.
740133            //
740134            inode->indirect1 = allocated_zone;
740135            inode->changed = 1;
740136            status = inode_save (inode);
740137            if (status != 0)
740138              {
740139                //
740140                // Cannot save the inode. This is an error and the value
740141                // for ‘errno’ is produced by ‘inode_save()’.
740142                //
740143                return ((zno_t) -1);
740144              }
740145          }
740146        //
740147        // An indirect table is present inside the file system:
740148        // load it.
740149        //
740150        status = zone_read (inode->sb, inode->indirect1, zone_table);
740151        if (status != 0)
740152          {
740153            //
740154            // Cannot load the indirect table. This is an error and the
740155            // value for ‘errno’ is assigned by function ‘zone_read()’.
740156            //
740157            return ((zno_t) -1);
740158          }
740159        //
740160        // The indirect table was read. Calculate the index inside
740161        // the table, for the requested zone.
740162        //
740163        i0 = (fzone - 7);
740164        //
740165        // Check if the zone is to be allocated.
740166        //
740167        if (zone_table[i0] == 0)
740168          {
740169            //
740170            // There is not such zone, but it is not an error.
740171            //
740172            if (!write)
740173              {
740174                return ((zno_t) 0);
740175              }
740176            //
740177            // The zone must be allocated.
740178            //
740179            allocated_zone = zone_alloc (inode->sb);
740180            if (allocated_zone == 0)
740181              {
740182                //
740183                // There is no space for the zone allocation. The
740184                // variable ‘errno’ is already updated by
740185                // ‘zone_alloc()’.
740186                //
740187                return ((zno_t) -1);
740188              }
740189            //
740190            // The zone is allocated: clear the zone and save.
740191            //
740192            memset (buffer, 0, SB_MAX_ZONE_SIZE);
740193            status = zone_write (inode->sb, allocated_zone, buffer);
```



```
740194            if (status < 0)
740195              {
740196                //
740197                // Cannot overwrite the zone. The variable ‘errno’ is
740198                // set by ‘zone_write()’.
740199                //
740200                return ((zno_t) -1);
740201              }
740202            //
740203            // The zone is allocated and cleared: update the indirect
740204            // zone table an save it. The inode is not modified,
740205            // because the indirect table is outside.
740206            //
740207            zone_table[i0] = allocated_zone;
740208            status = zone_write (inode->sb, inode->indirect1, zone_table);
740209            if (status != 0)
740210              {
740211                //
740212                // Cannot save the zone. The variable ‘errno’ is already
740213                // set by ‘zone_write()’.
740214                //
740215                return ((zno_t) -1);
740216              }
740217          }
740218        //
740219        // The zone is allocated.
740220        //
740221        return (zone_table[i0]);
740222      }
740223    else
740224      {
740225        //
740226        // (7 + indirect_zones) <= fzone
740227        // The zone number is inside the double indirect zone
740228        // references.
740229        // Verify to have the first level of second indirection.
740230        //
740231        if (inode->indirect2 == 0)
740232          {
740233            //
740234            // There is not such zone, but it is not an error.
740235            //
740236            if (!write)
740237              {
740238                return ((zno_t) 0);
740239              }
740240            //
740241            // The first level of second indirection must be
740242            // initialized.
740243            //
740244            allocated_zone = zone_alloc (inode->sb);
740245            if (allocated_zone == 0)
740246              {
740247                //
740248                // Cannot allocate the zone. The variable ‘errno’ is
740249                // set by ‘zone_alloc()’.
740250                //
740251                return ((zno_t) -1);
740252              }
740253            //
740254            // The zone for the indirection table is allocated:
740255            // clear the zone and save.
740256            //
740257            memset (buffer, 0, SB_MAX_ZONE_SIZE);
740258            status = zone_write (inode->sb, allocated_zone, buffer);
740259            if (status < 0)
740260              {
740261                //
740262                // Cannot overwrite the zone. The variable ‘errno’ is
740263                // set by ‘zone_write()’.
740264                //
740265                return ((zno_t) -1);
740266              }
740267            //
740268            // The zone for the indirection table is allocated and
740269            // cleared: save the inode.
740270            //
740271            inode->indirect2 = allocated_zone;
740272            inode->changed = 1;
740273            status = inode_save (inode);
740274            if (status != 0)
740275              {
740276                //
740277                // Cannot save the inode. The variable ‘errno’ is
740278                // set by ‘inode_save()’.
740279                //
740280                return ((zno_t) -1);
740281              }
740282          }
740283        //
740284        // The first level of second indirection is present:
740285        // Read the second indirect table.
740286        //
740287        status = zone_read (inode->sb, inode->indirect2, zone_table);
740288        if (status != 0)
740289          {
740290            //
740291            // Cannot read the second indirect table. The variable
740292            // ‘errno’ is set by ‘zone_read()’.
740293            //
740294            return ((zno_t) -1);
```

```
740295          }
740296        //
740297        // The first double indirect table was read: calculate
740298        // indexes inside first and second level of table.
740299        //
740300        fzone -= 7;
740301        fzone -= indirect_zones;
740302        i1     = fzone / indirect_zones;
740303        i2     = fzone % indirect_zones;
740304        //
740305        // Verify to have a second level.
740306        //
740307        if (zone_table[i1] == 0)
740308          {
740309            //
740310            // There is not such zone, but it is not an error.
740311            //
740312            if (!write)
740313              {
740314                return ((zno_t) 0);
740315              }
740316            //
740317            // The second level must be initialized.
740318            //
740319            allocated_zone = zone_alloc (inode->sb);
740320            if (allocated_zone == 0)
740321              {
740322                //
740323                // Cannot allocate the zone. The variable ‘errno’ is set
740324                // by ‘zone_alloc()’.
740325                //
740326                return ((zno_t) -1);
740327              }
740328            //
740329            // The zone for the indirection table is allocated:
740330            // clear the zone and save.
740331            //
740332            memset (buffer, 0, SB_MAX_ZONE_SIZE);
740333            status = zone_write (inode->sb, allocated_zone, buffer);
740334            if (status < 0)
740335              {
740336                //
740337                // Cannot overwrite the zone. The variable ‘errno’ is
740338                // set by ‘zone_write()’.
740339                //
740340                return ((zno_t) -1);
740341              }
740342            //
740343            // Update the first level index and save it.
740344            //
740345            zone_table[i1] = allocated_zone;
740346            status = zone_write (inode->sb, inode->indirect2, zone_table);
740347            if (status != 0)
740348              {
740349                //
740350                // Cannot write the zone. The variable ‘errno’ is set
740351                // by ‘zone_write()’.
740352                //
740353                return ((zno_t) -1);
740354              }
740355          }
740356        //
740357        // The second level can be read, overwriting the array
740358        // ‘zone_table[]’. The zone number for the second level
740359        // indirection table is saved inside ‘zone_second’, before
740360        // overwriting the array.
740361        //
740362        zone_second = zone_table[i1];
740363        status = zone_read (inode->sb, zone_second, zone_table);
740364        if (status != 0)
740365          {
740366            //
740367            // Cannot read the second level indirect table. The variable
740368            // ‘errno’ is set by ‘zone_read()’.
740369            //
740370            return ((zno_t) -1);
740371          }
740372        //
740373        // The second level was read and ‘zone_table[]’ is now
740374        // such second one: check if the zone is to be allocated.
740375        //
740376        if (zone_table[i2] == 0)
740377          {
740378            //
740379            // There is not such zone, but it is not an error.
740380            //
740381            if (!write)
740382              {
740383                return ((zno_t) 0);
740384              }
740385            //
740386            // Must be allocated.
740387            //
740388            allocated_zone = zone_alloc (inode->sb);
740389            if (allocated_zone == 0)
740390              {
740391                //
740392                // Cannot allocate the zone. The variable ‘errno’ is set
740393                // by ‘zone_alloc()’.
740394                //
740395                return ((zno_t) -1);
```



```
740396              }
740397            //
740398            // The zone is allocated: clear the zone and save.
740399            //
740400            memset (buffer, 0, SB_MAX_ZONE_SIZE);
740401            status = zone_write (inode->sb, allocated_zone, buffer);
740402            if (status < 0)
740403              {
740404                //
740405                // Cannot overwrite the zone. The variable ‘errno’ is
740406                // set by ‘zone_write()’.
740407                //
740408                return ((zno_t) -1);
740409              }
740410            //
740411            // The zone was allocated and cleared: update the indirect
740412            // zone table an save it. The inode is not modified, because
740413            // the indirect table is outside.
740414            //
740415            zone_table[i2] = allocated_zone;
740416            status = zone_write (inode->sb, zone_second, zone_table);
740417            if (status != 0)
740418              {
740419                //
740420                // Cannot write the zone. The variable ‘errno’ is set
740421                // by ‘zone_write()’.
740422                //
740423                return ((zno_t) -1);
740424              }
740425          }
740426        //
740427        // The zone is there: return the zone number.
740428        //
740429        return (zone_table[i2]);
740430      }
740431  }
```

## kernel/fs/path_chdir.c

«

Si veda la sezione i159.3.30.

```
750001  #include <kernel/fs.h>
750002  #include <errno.h>
750003  #include <kernel/proc.h>
750004  //------------------------------------------------------------------------
750005  int
750006  path_chdir (pid_t pid, const char *path)
750007  {
750008      proc_t   *ps;
750009      inode_t  *inode_directory;
750010      int       status;
750011      char      path_directory[PATH_MAX];
750012      //
750013      // Get process.
750014      //
750015      ps = proc_reference (pid);
750016      //
750017      // The full directory path is needed.
750018      //
750019      status = path_full (path, ps->path_cwd, path_directory);
750020      if (status < 0)
750021        {
750022          return (-1);
750023        }
750024      //
750025      // Try to load the new directory inode.
750026      //
750027      inode_directory = path_inode (pid, path_directory);
750028      if (inode_directory == NULL)
750029        {
750030          //
750031          // Cannot access the directory: it does not exists or
750032          // permissions are not sufficient. Variable ‘errno’ is set by
750033          // function ‘inode_directory()’.
750034          //
750035          errset (errno);
750036          return (-1);
750037        }
750038      //
750039      // Inode loaded: release the old directory and set the new one.
750040      //
750041      inode_put (ps->inode_cwd);
750042      //
750043      ps->inode_cwd = inode_directory;
750044      strncpy (ps->path_cwd, path_directory, PATH_MAX);
750045      //
750046      // Return.
750047      //
750048      return (0);
750049  }
```

## kernel/fs/path_chmod.c

«

Si veda la sezione i159.3.31.

```
760001  #include <kernel/fs.h>
760002  #include <errno.h>
760003  #include <kernel/proc.h>
760004  //------------------------------------------------------------------------
760005  int
```

```
760006  path_chmod (pid_t pid, const char *path, mode_t mode)
760007  {
760008      proc_t   *ps;
760009      inode_t  *inode;
760010      //
760011      // Get process.
760012      //
760013      ps = proc_reference (pid);
760014      //
760015      // Try to load the file inode.
760016      //
760017      inode = path_inode (pid, path);
760018      if (inode == NULL)
760019        {
760020          //
760021          // Cannot access the file: it does not exists or permissions are
760022          // not sufficient. Variable ‘errno’ is set by function
760023          // ‘inode_directory()’.
760024          //
760025          return (-1);
760026        }
760027      //
760028      // Verify to be root or to be the owner.
760029      //
760030      if (ps->euid != 0 && ps->euid != inode->uid)
760031        {
760032          errset (EACCES);        // Permission denied.
760033          return (-1);
760034        }
760035      //
760036      // Update the mode: the file type is kept and the
760037      // rest is taken form the parameter ‘mode’.
760038      //
760039      inode->mode = (S_IFMT & inode->mode) | (~S_IFMT & mode);
760040      //
760041      // Save and release the inode.
760042      //
760043      inode->changed = 1;
760044      inode_save (inode);
760045      inode_put (inode);
760046      //
760047      // Return.
760048      //
760049      return (0);
760050  }
```

## kernel/fs/path_chown.c

«

Si veda la sezione i159.3.32.

```
770001  #include <kernel/fs.h>
770002  #include <errno.h>
770003  #include <kernel/proc.h>
770004  //-------------------------------------------------------------------------
770005  int
770006  path_chown (pid_t pid, const char *path, uid_t uid, gid_t gid)
770007  {
770008      proc_t   *ps;
770009      inode_t  *inode;
770010      //
770011      // Get process.
770012      //
770013      ps = proc_reference (pid);
770014      //
770015      // Must be root, as the ability to change group is not considered.
770016      //
770017      if (ps->euid != 0)
770018        {
770019          errset (EPERM);                 // Operation not permitted.
770020          return (-1);
770021        }
770022      //
770023      // Try to load the file inode.
770024      //
770025      inode = path_inode (pid, path);
770026      if (inode == NULL)
770027        {
770028          //
770029          // Cannot access the file: it does not exists or permissions are
770030          // not sufficient. Variable ‘errno’ is set by function
770031          // ‘inode_directory()’.
770032          //
770033          return (-1);
770034        }
770035      //
770036      // Update the owner and group.
770037      //
770038      if (uid != -1)
770039        {
770040          inode->uid     = uid;
770041          inode->changed = 1;
770042        }
770043      if (gid != -1)
770044        {
770045          inode->gid     = gid;
770046          inode->changed = 1;
770047        }
770048      //
770049      // Save and release the inode.
770050      //
770051      inode_save (inode);
```



```
770052      inode_put (inode);
770053      //
770054      // Return.
770055      //
770056      return (0);
770057  }
```

## kernel/fs/path_device.c

«

Si veda la sezione i159.3.33.

```
780001  #include <kernel/fs.h>
780002  #include <errno.h>
780003  #include <kernel/proc.h>
780004  //-------------------------------------------------------------------------
780005  dev_t
780006  path_device (pid_t pid, const char *path)
780007  {
780008      proc_t   *ps;
780009      inode_t  *inode;
780010      dev_t     device;
780011      //
780012      // Get process.
780013      //
780014      ps = proc_reference (pid);
780015      //
780016      inode = path_inode (pid, path);
780017      if (inode == NULL)
780018        {
780019          errset (errno);
780020          return ((dev_t) -1);
780021        }
780022      //
780023      if (!(S_ISBLK (inode->mode) || S_ISCHR (inode->mode)))
780024        {
780025          errset (ENODEV);                // No such device.
780026          inode_put (inode);
780027          return ((dev_t) -1);
780028        }
780029      //
780030      device = inode->direct[0];
780031      inode_put (inode);
780032      return (device);
780033  }
```

## kernel/fs/path_fix.c

«

Si veda la sezione i159.3.34.

```
790001  #include <kernel/fs.h>
790002  #include <errno.h>
790003  #include <kernel/proc.h>
790004  //-------------------------------------------------------------------------
790005  int
790006  path_fix (char *path)
790007  {
790008      char    new_path[PATH_MAX];
790009      char   *token[PATH_MAX/4];
790010      int     t;                          // Token index.
790011      int     token_size;                 // Token array effective size.
790012      int     comp;                       // String compare return value.
790013      size_t  path_size;                  // Path string size.
790014      //
790015      // Initialize token search.
790016      //
790017      token[0] = strtok (path, "/");
790018      //
790019      // Scan tokens.
790020      //
790021      for (t = 0;
790022           t < PATH_MAX/4 && token[t] != NULL;
790023           t++, token[t] = strtok (NULL, "/"))
790024        {
790025          //
790026          // If current token is ‘.’, just ignore it.
790027          //
790028          comp = strcmp (token[t], ".");
790029          if (comp == 0)
790030            {
790031              t--;
790032            }
790033          //
790034          // If current token is ‘..’, remove previous token,
790035          // if there is one.
790036          //
790037          comp = strcmp (token[t], "..");
790038          if (comp == 0)
790039            {
790040              if (t > 0)
790041                {
790042                  t -= 2;
790043                }
790044              else
790045                {
790046                  t = -1;
790047                }
790048            }
790049          //
790050          // ‘t’ will be incremented and another token will be
790051          // found.
```

```
790052          //
790053        }
790054      //
790055      // Save the token array effective size.
790056      //
790057      token_size = t;
790058      //
790059      // Initialize the new path string.
790060      //
790061      new_path[0] = '\0';
790062      //
790063      // Build the new path string.
790064      //
790065      if (token_size > 0)
790066        {
790067          for (t = 0; t < token_size; t++)
790068            {
790069              path_size = strlen (new_path);
790070              strncat (new_path, "/", 2);
790071              strncat (new_path, token[t], PATH_MAX - path_size - 1);
790072            }
790073        }
790074      else
790075        {
790076          strncat (new_path, "/", 2);
790077        }
790078      //
790079      // Copy the new path into the original string.
790080      //
790081      strncpy (path, new_path, PATH_MAX);
790082      //
790083      // Return.
790084      //
790085      return (0);
790086  }
```

kernel/fs/path_full.c

«

Si veda la sezione i159.3.35.

```
800001  #include <kernel/fs.h>
800002  #include <errno.h>
800003  #include <kernel/proc.h>
800004  //-------------------------------------------------------------------------
800005  int
800006  path_full (const char *path, const char *path_cwd, char *full_path)
800007  {
800008      unsigned int path_size;
800009      //
800010      // Check some arguments.
800011      //
800012      if (path == NULL || strlen (path) == 0 || full_path == NULL)
800013        {
800014          errset (EINVAL);                // Invalid argument.
800015          return (-1);
800016        }
800017      //
800018      // The main path and the receiving one are right.
800019      // Now arrange to get a full path name.
800020      //
800021      if (path[0] == '/')
800022        {
800023          strncpy (full_path, path, PATH_MAX);
800024          full_path[PATH_MAX-1] = 0;
800025        }
800026      else
800027        {
800028          if (path_cwd == NULL || strlen (path_cwd) == 0)
800029            {
800030              errset (EINVAL);            // Invalid argument.
800031              return (-1);
800032            }
800033          strncpy (full_path, path_cwd, PATH_MAX);
800034          path_size = strlen (full_path);
800035          strncat (full_path, "/", (PATH_MAX - path_size));
800036          path_size = strlen (full_path);
800037          strncat (full_path, path, (PATH_MAX - path_size));
800038        }
800039      //
800040      // Fix path name so that it has no '..', '.', and no
800041      // multiple '/'.
800042      //
800043      path_fix (full_path);
800044      //
800045      // Return.
800046      //
800047      return (0);
800048  }
```

kernel/fs/path_inode.c

«

Si veda la sezione i159.3.36.

```
810001  #include <kernel/fs.h>
810002  #include <errno.h>
810003  #include <kernel/proc.h>
810004  #include <kernel/k_libc.h>
810005  //-------------------------------------------------------------------------
810006  #define DIRECTORY_BUFFER_SIZE (SB_MAX_ZONE_SIZE/16)
810007  //-------------------------------------------------------------------------
```



```
810008  inode_t *
810009  path_inode (pid_t pid, const char *path)
810010  {
810011      proc_t     *ps;
810012      inode_t    *inode;
810013      dev_t       device;
810014      char        full_path[PATH_MAX];
810015      char       *name;
810016      char       *next;
810017      directory_t dir[DIRECTORY_BUFFER_SIZE];
810018      char        dir_name[NAME_MAX+1];
810019      off_t       offset_dir;
810020      ssize_t     size_read;
810021      size_t      dir_size_read;
810022      ssize_t     size_to_read;
810023      int         comp;
810024      int         d;              // Directory index;
810025      int         status;         // inode_check() return status.
810026      //
810027      // Get process.
810028      //
810029      ps = proc_reference (pid);
810030      //
810031      // Arrange to get a packed full path name.
810032      //
810033      path_full (path, ps->path_cwd, full_path);
810034      //
810035      // Get the root file system inode.
810036      //
810037      inode = inode_get ((dev_t) 0, 1);
810038      if (inode == NULL)
810039        {
810040          errset (errno);
810041          return (NULL);
810042        }
810043      //
810044      // Save the device number.
810045      //
810046      device = inode->sb->device;
810047      //
810048      // Variable 'inode' already points to the root file system inode:
810049      // It must be a directory!
810050      //
810051      status = inode_check (inode, S_IFDIR, 1, ps->euid);
810052      if (status != 0)
810053        {
810054          //
810055          // Variable 'errno' should be set by inode_check().
810056          //
810057          errset (errno);
810058          inode_put (inode);
810059          return (NULL);
810060        }
810061      //
810062      // Initialize string scan: find the first path token, after the
810063      // first '/'.
810064      //
810065      name = strtok (full_path, "/");
810066      //
810067      // If the original full path is just '/' the variable 'name'
810068      // appears as a null pointer, and the variable 'inode' is already
810069      // what we are looking for.
810070      //
810071      if (name == NULL)
810072        {
810073          return (inode);
810074        }
810075      //
810076      // There is at least a name after '/' inside the original full
810077      // path. A scan is going to start: the original value for variable
810078      // 'inode' is a pointer to the root directory inode.
810079      //
810080      for (;;)
810081        {
810082          //
810083          // Find next token.
810084          //
810085          next = strtok (NULL, "/");
810086          //
810087          // Read the directory from the current inode.
810088          //
810089          for (offset_dir=0; ; offset_dir += size_read)
810090            {
810091              size_to_read = DIRECTORY_BUFFER_SIZE;
810092              //
810093              if ((offset_dir + size_to_read) > inode->size)
810094                {
810095                  size_to_read = inode->size - offset_dir;
810096                }
810097              //
810098              size_read = inode_file_read (inode, offset_dir, dir,
810099                                           size_to_read, NULL);
810100              //
810101              // The size read must be a multiple of 16.
810102              //
810103              size_read = ((size_read / 16) * 16);
810104              //
810105              // Check anyway if it is zero.
810106              //
810107              if (size_read == 0)
810108                {
```

```
                //
                // The directory is ended: release the inode and return.
                //
                inode_put (inode);
                errset (ENOENT);           // No such file or directory.
                return (NULL);
              }
            //
            // Calculate how many directory items we have read.
            //
            dir_size_read = size_read / 16;
            //
            // Scan the directory to find the current name.
            //
            for (d = 0; d < dir_size_read; d++)
              {
                //
                // Ensure to have a null terminated string for
                // the name found.
                //
                memcpy (dir_name, dir[d].name, (size_t) NAME_MAX);
                dir_name[NAME_MAX] = 0;
                //
                comp = strcmp (name, dir_name);
                if (comp == 0 && dir[d].ino != 0)
                  {
                    //
                    // Found the name and verified that it has a link to
                    // a inode. Now release the directory inode.
                    //
                    inode_put (inode);
                    //
                    // Get next inode and break the loop.
                    //
                    inode = inode_get (device, dir[d].ino);
                    break;
                  }
              }
            //
            // If index 'd' is in a valid range, the name was found.
            //
            if (d < dir_size_read)
              {
                //
                // The name was found.
                //
                break;
              }
          }
        //
        // If the function is still working, a file or a directory
        // was found: see if there is another name after this one
        // to look for. If there isn't, just break the loop.
        //
        if (next == NULL)
          {
            //
            // As no other tokens are to be found, break the loop.
            //
            break;
          }
        //
        // As there is another name after the current one,
        // the current file must be a directory.
        //
        status = inode_check (inode, S_IFDIR, 1, ps->euid);
        if (status != 0)
          {
            //
            // Variable 'errno' is set by 'inode_check()'.
            //
            errset (errno);
            inode_put (inode);
            return (NULL);
         }
        //
        // The inode is a directory and the user has the necessary
        // permissions: check if it is a mount point and go to the
        // new device root directory if necessary.
        //
        if (inode->sb_attached != NULL)
          {
            //
            // Must find the root directory for the new device, and
            // then go to that inode.
            //
            device = inode->sb_attached->device;
            inode_put (inode);
            inode = inode_get (device, 1);
            status = inode_check (inode, S_IFDIR, 1, ps->euid);
            if (status != 0)
              {
                inode_put (inode);
                return (NULL);
              }
          }
        //
        // As a directory was found, and another token follows it,
        // must continue the token scan.
        //
        name = next;
```



```
      }
    //
    // Current inode found is the file represented by the requested
    // path.
    //
    return (inode);
}
```

## kernel/fs/path_inode_link.c

«

Si veda la sezione i159.3.37.

```
#include <kernel/fs.h>
#include <errno.h>
#include <kernel/proc.h>
#include <libgen.h>
//-----------------------------------------------------------------------
inode_t *
path_inode_link (pid_t pid, const char *path, inode_t *inode,
                 mode_t mode)
{
    proc_t      *ps;
    char         buffer[SB_MAX_ZONE_SIZE];
    off_t        start;
    int          d;       // Directory index.
    ssize_t      size_read;
    ssize_t      size_written;
    directory_t *dir = (directory_t *) buffer;
    char         path_copy1[PATH_MAX];
    char         path_copy2[PATH_MAX];
    char        *path_directory;
    char        *path_name;
    inode_t     *inode_directory;
    inode_t     *inode_new;
    dev_t        device;
    int          status;
    //
    // Check arguments.
    //
    if (path == NULL || strlen (path) == 0)
      {
        errset (EINVAL);        // Invalid argument:
        return (NULL);          // the path is mandatory.
      }
    //
    if (inode == NULL && mode == 0)
      {
        errset (EINVAL);        // Invalid argument: if the inode is to
        return (NULL);          // be created, the mode is mandatory.
      }
    //
    if (inode != NULL)
      {
        if (mode != 0)
          {
            errset (EINVAL);    // Invalid argument: if the inode is
            return (NULL);      // already present, the creation mode
          }                     // must not be given.
        if (S_ISDIR (inode->mode))
          {
            errset (EPERM);     // Operation not permitted.
            return (NULL);      // Refuse to link directory.
          }
        if (inode->links >= LINK_MAX)
          {
            errset (EMLINK);            // Too many links.
            return (NULL);
          }
      }
    //
    // Get process.
    //
    ps = proc_reference (pid);
    //
    // If the destination path already exists, the link cannot be made.
    // It does not matter if the inode is known or not.
    //
    inode_new = path_inode ((uid_t) 0, path);
    if (inode_new != NULL)
      {
        //
        // A file already exists with the same name.
        //
        inode_put (inode_new);
        errset (EEXIST);                // File exists.
        return (NULL);
      }
    //
    // At this point, 'inode_new' is 'NULL'.
    // Copy the source path inside the directory path and name arrays.
    //
    strncpy (path_copy1, path, PATH_MAX);
    strncpy (path_copy2, path, PATH_MAX);
    //
    // Reduce to directory name and find the last name.
    //
    path_directory = dirname  (path_copy1);
    path_name      = basename (path_copy2);
    if (strlen (path_directory) == 0 || strlen (path_name) == 0)
      {
        errset (EACCES);        // Permission denied: maybe the
```

```
820090                                  // original path is the root directory
820091                                  // and cannot find a previous directory.
820092            return (NULL);
820093          }
820094        //
820095        // Get the directory inode.
820096        //
820097        inode_directory = path_inode (pid, path_directory);
820098        if (inode_directory == NULL)
820099          {
820100            errset (errno);
820101            return (NULL);
820102          }
820103        //
820104        // Check if something is mounted on it.
820105        //
820106        if (inode_directory->sb_attached != NULL)
820107          {
820108            //
820109            // Must select the right directory.
820110            //
820111            device = inode_directory->sb_attached->device;
820112            inode_put (inode_directory);
820113            inode_directory = inode_get (device, 1);
820114            if (inode_directory == NULL)
820115              {
820116                return (NULL);
820117              }
820118          }
820119        //
820120        // If the inode to link is known, check if the selected directory
820121        // has the same super block than the inode to link.
820122        //
820123        if (inode != NULL && inode_directory->sb != inode->sb)
820124          {
820125            inode_put (inode_directory);
820126            errset (ENOENT);        // No such file or directory.
820127            return (NULL);
820128          }
820129        //
820130        // Check if write is allowed for the file system.
820131        //
820132        if (inode_directory->sb->options & MOUNT_RO)
820133          {
820134            inode_put (inode_directory);
820135            errset (EROFS);         // Read-only file system.
820136            return (NULL);
820137          }
820138        //
820139        // Verify access permissions for the directory. The number "3" means
820140        // that the user must have access permission and write permission:
820141        // "-wx" == 2+1 == 3.
820142        //
820143        status = inode_check (inode_directory, S_IFDIR, 3, ps->euid);
820144        if (status != 0)
820145          {
820146            inode_put (inode_directory);
820147            return (NULL);
820148          }
820149        //
820150        // If the inode to link was not specified, it must be created.
820151        // From now on, the inode is referenced with the variable
820152        // 'inode_new'.
820153        //
820154        inode_new = inode;
820155        //
820156        if (inode_new == NULL)
820157          {
820158            inode_new = inode_alloc (inode_directory->sb->device, mode,
820159                                     ps->euid);
820160            if (inode_new == NULL)
820161              {
820162                //
820163                // The inode allocation failed, so, also the directory
820164                // must be released, before return.
820165                //
820166                inode_put (inode_directory);
820167                return (NULL);
820168              }
820169          }
820170        //
820171        // Read the directory content and try to add the new item.
820172        //
820173        for (start  = 0;
820174             start <  inode_directory->size;
820175             start += inode_directory->sb->blksize)
820176          {
820177            size_read = inode_file_read (inode_directory, start, buffer,
820178                                         inode_directory->sb->blksize,
820179                                         NULL);
820180            if (size_read < sizeof (directory_t))
820181              {
820182                break;
820183              }
820184            //
820185            // Scan the directory portion just read, for an unused item.
820186            //
820187            dir = (directory_t *) buffer;
820188            for (d = 0; d < size_read; d += (sizeof (directory_t)), dir++)
820189              {
820190                if (dir->ino == 0)
```



```
820191                  {
820192                    //
820193                    // Found an empty directory item: link the inode.
820194                    //
820195                    dir->ino = inode_new->ino;
820196                    strncpy (dir->name, path_name, NAME_MAX);
820197                    inode_new->links++;
820198                    inode_new->changed = 1;
820199                    //
820200                    // Update the directory inside the file system.
820201                    //
820202                    size_written = inode_file_write (inode_directory, start,
820203                                                     buffer, size_read);
820204                    if (size_written != size_read)
820205                      {
820206                        //
820207                        // Write problem: release the directory and return.
820208                        //
820209                        inode_put (inode_directory);
820210                        errset (EUNKNOWN);
820211                        return (NULL);
820212                      }
820213                    //
820214                    // Save the new inode, release the directory and return
820215                    // the linked inode.
820216                    //
820217                    inode_save (inode_new);
820218                    inode_put (inode_directory);
820219                    return (inode_new);
820220                  }
820221              }
820222          }
820223        //
820224        // The directory don't have a free item and one must be appended.
820225        //
820226        dir   = (directory_t *) buffer;
820227        start = inode_directory->size;
820228        //
820229        // Prepare the buffer with the link.
820230        //
820231        dir->ino = inode_new->ino;
820232        strncpy (dir->name, path_name, NAME_MAX);
820233        inode_new->links++;
820234        inode_new->changed = 1;
820235        //
820236        // Append the buffer to the directory.
820237        //
820238        size_written = inode_file_write (inode_directory, start, buffer,
820239                                         (sizeof (directory_t)));
820240        if (size_written != (sizeof (directory_t)))
820241          {
820242            //
820243            // Problem updating the directory: release it and return.
820244            //
820245            inode_put (inode_directory);
820246            errset (EUNKNOWN);
820247            return (NULL);
820248          }
820249        //
820250        // Close access to the directory inode and save the other inode,
820251        // with updated link count.
820252        //
820253        inode_put (inode_directory);
820254        inode_save (inode_new);
820255        //
820256        // Return successfully.
820257        //
820258        return (inode_new);
820259    }
```

## kernel/fs/path_link.c

«

Si veda la sezione i159.3.38.

```
830001  #include <kernel/fs.h>
830002  #include <errno.h>
830003  #include <kernel/proc.h>
830004  //----------------------------------------------------------------------
830005  int
830006  path_link (pid_t pid, const char *path_old, const char *path_new)
830007  {
830008      proc_t             *ps;
830009      inode_t            *inode_old;
830010      inode_t            *inode_new;
830011      char                path_new_full[PATH_MAX];
830012      //
830013      // Get process.
830014      //
830015      ps = proc_reference (pid);
830016      //
830017      // Try to get the old path inode.
830018      //
830019      inode_old = path_inode (pid, path_old);
830020      if (inode_old == NULL)
830021        {
830022          //
830023          // Cannot get the inode: 'errno' is already set by
830024          // 'path_inode()'.
830025          //
830026          errset (errno);
830027          return (-1);
```

```
830028        }
830029      //
830030      // The inode is available and checks are done: arrange to get a
830031      // packed full path name and then the destination directory path.
830032      //
830033      path_full (path_new, ps->path_cwd, path_new_full);
830034      //
830035      //
830036      //
830037      inode_new = path_inode_link (pid, path_new_full, inode_old,
830038                                   (mode_t) 0);
830039      if (inode_new == NULL)
830040        {
830041          inode_put (inode_old);
830042          return (-1);
830043        }
830044      if (inode_new != inode_old)
830045        {
830046          inode_put (inode_new);
830047          inode_put (inode_old);
830048          errset (EUNKNOWN);              // Unknown error.
830049          return (-1);
830050        }
830051      //
830052      // Inode data is already updated by 'path_inode_link()': just put
830053      // it and return. Please note that only one is put, because it is
830054      // just the same of the other.
830055      //
830056      inode_put  (inode_new);
830057      return (0);
830058  }
```

kernel/fs/path_mkdir.c

«

Si veda la sezione i159.3.39.

```
840001  #include <kernel/fs.h>
840002  #include <errno.h>
840003  #include <kernel/proc.h>
840004  #include <libgen.h>
840005  #include <kernel/k_libc.h>
840006  //-------------------------------------------------------------------------
840007  int
840008  path_mkdir (pid_t pid, const char *path, mode_t mode)
840009  {
840010      proc_t   *ps;
840011      inode_t  *inode_directory;
840012      inode_t  *inode_parent;
840013      int       status;
840014      char      path_directory[PATH_MAX];
840015      char      path_copy[PATH_MAX];
840016      char     *path_parent;
840017      ssize_t   size_written;
840018      //
840019      struct {
840020          ino_t inode_1;
840021          char  name_1[NAME_MAX];
840022          ino_t inode_2;
840023          char  name_2[NAME_MAX];
840024      } directory;
840025      //
840026      // Get process.
840027      //
840028      ps = proc_reference (pid);
840029      //
840030      // Correct the mode with the umask.
840031      //
840032      mode &= ~ps->umask;
840033      //
840034      // Inside 'mode', the file type is fixed. No check is made.
840035      //
840036      mode &= 00777;
840037      mode |= S_IFDIR;
840038      //
840039      // The full path and the directory path is needed.
840040      //
840041      status = path_full (path, ps->path_cwd, path_directory);
840042      if (status < 0)
840043        {
840044          return (-1);
840045        }
840046      strncpy (path_copy, path_directory, PATH_MAX);
840047      path_copy[PATH_MAX-1] = 0;
840048      path_parent = dirname (path_copy);
840049      //
840050      // Check if something already exists with the same name. The scan
840051      // is done with kernel privileges.
840052      //
840053      inode_directory = path_inode ((uid_t) 0, path_directory);
840054      if (inode_directory != NULL)
840055        {
840056          //
840057          // The file already exists. Put inode and return an error.
840058          //
840059          inode_put (inode_directory);
840060          errset (EEXIST);                // File exists.
840061          return (-1);
840062        }
840063      //
840064      // Try to locate the directory that should contain this one.
840065      //
```



```
840066      inode_parent = path_inode (pid, path_parent);
840067      if (inode_parent == NULL)
840068        {
840069          //
840070          // Cannot locate the directory: return an error. The variable
840071          // 'errno' should already be set by 'path_inode()'.
840072          //
840073          errset (errno);
840074          return (-1);
840075        }
840076      //
840077      // Try to create the node: should fail if the user does not have
840078      // enough permissions.
840079      //
840080      inode_directory = path_inode_link (pid, path_directory, NULL,
840081                                         mode);
840082      if (inode_directory == NULL)
840083        {
840084          //
840085          // Sorry: cannot create the inode! The variable 'errno' should
840086          // already be set by 'path_inode_link()'.
840087          //
840088          errset (errno);
840089          return (-1);
840090        }
840091      //
840092      // Fill records for '.' and '..'.
840093      //
840094      directory.inode_1 = inode_directory->ino;
840095      strncpy (directory.name_1, ".",  (size_t) 3);
840096      directory.inode_2 = inode_parent->ino;
840097      strncpy (directory.name_2, "..", (size_t) 3);
840098      //
840099      // Write data.
840100      //
840101      size_written = inode_file_write (inode_directory, (off_t) 0,
840102                                       &directory, (sizeof directory));
840103      if (size_written != (sizeof directory))
840104        {
840105          return (-1);
840106        }
840107      //
840108      // Fix directory inode links.
840109      //
840110      inode_directory->links   = 2;
840111      inode_directory->time    = k_time (NULL);
840112      inode_directory->changed = 1;
840113      //
840114      // Fix parent directory inode links.
840115      //
840116      inode_parent->links++;
840117      inode_parent->time       = k_time (NULL);
840118      inode_parent->changed    = 1;
840119      //
840120      // Save and put the inodes.
840121      //
840122      inode_save (inode_parent);
840123      inode_save (inode_directory);
840124      inode_put  (inode_parent);
840125      inode_put  (inode_directory);
840126      //
840127      // Return.
840128      //
840129      return (0);
840130  }
```

kernel/fs/path_mknod.c

«

Si veda la sezione i159.3.40.

```
850001  #include <kernel/fs.h>
850002  #include <errno.h>
850003  #include <kernel/proc.h>
850004  //-------------------------------------------------------------------------
850005  int
850006  path_mknod (pid_t pid, const char *path, mode_t mode, dev_t device)
850007  {
850008      proc_t   *ps;
850009      inode_t  *inode;
850010      char      full_path[PATH_MAX];
850011      //
850012      // Get process.
850013      //
850014      ps = proc_reference (pid);
850015      //
850016      // Correct the mode with the umask.
850017      //
850018      mode &= ~ps->umask;
850019      //
850020      // Currently must be root for any kind of node to be created.
850021      //
850022      if (ps->uid != 0)
850023        {
850024          errset (EPERM);                 // Operation not permitted.
850025          return (-1);
850026        }
850027      //
850028      // Check the type of node requested.
850029      //
850030      if (!(S_ISBLK (mode) ||
850031            S_ISCHR (mode) ||
```

```
850032            S_ISREG (mode) ||
850033            S_ISDIR (mode)))
850034        {
850035          errset (EINVAL);                // Invalid argument.
850036          return (-1);
850037        }
850038      //
850039      // Check if something already exists with the same name.
850040      //
850041      inode = path_inode (pid, path);
850042      if (inode != NULL)
850043        {
850044          //
850045          // The file already exists. Put inode and return an error.
850046          //
850047          inode_put (inode);
850048          errset (EEXIST);                // File exists.
850049          return (-1);
850050        }
850051      //
850052      // Try to creat the node.
850053      //
850054      path_full (path, ps->path_cwd, full_path);
850055      inode = path_inode_link (pid, full_path, NULL, mode);
850056      if (inode == NULL)
850057        {
850058          //
850059          // Sorry: cannot create the inode!
850060          //
850061          return (-1);
850062        }
850063      //
850064      // Set the device number if necessary.
850065      //
850066      if (S_ISBLK (mode) || S_ISCHR (mode))
850067        {
850068          inode->direct[0] = device;
850069          inode->changed = 1;
850070        }
850071      //
850072      // Put the inode.
850073      //
850074      inode_put (inode);
850075      //
850076      // Return.
850077      //
850078      return (0);
850079  }
```

kernel/fs/path_mount.c

«

Si veda la sezione i159.3.41.

```
860001  #include <kernel/fs.h>
860002  #include <errno.h>
860003  #include <kernel/proc.h>
860004  //-----------------------------------------------------------------------
860005  int
860006  path_mount (pid_t pid, const char *path_dev, const char *path_mnt,
860007              int options)
860008  {
860009      proc_t   *ps;
860010      dev_t     device;        // Device to mount.
860011      inode_t  *inode_mnt;     // Directory mount point.
860012      void     *pstatus;
860013      //
860014      // Get process.
860015      //
860016      ps = proc_reference (pid);
860017      //
860018      // Verify to be the super user.
860019      //
860020      if (ps->euid != 0)
860021        {
860022          errset (EPERM);         // Operation not permitted.
860023          return (-1);
860024        }
860025      //
860026      device = path_device (pid, path_dev);
860027      if (device < 0)
860028        {
860029          return (-1);
860030        }
860031      //
860032      inode_mnt = path_inode (pid, path_mnt);
860033      if (inode_mnt == NULL)
860034        {
860035          return (-1);
860036        }
860037      if (!S_ISDIR (inode_mnt->mode))
860038        {
860039          inode_put (inode_mnt);
860040          errset (ENOTDIR);       // Not a directory.
860041          return (-1);
860042        }
860043      if (inode_mnt->sb_attached != NULL)
860044        {
860045          inode_put (inode_mnt);
860046          errset (EBUSY);         // Device or resource busy.
860047          return (-1);
860048        }
```



```
860049      //
860050      // All data is available.
860051      //
860052      pstatus = sb_mount (device, &inode_mnt, options);
860053      if (pstatus == NULL)
860054        {
860055          inode_put (inode_mnt);
860056          return (-1);
860057        }
860058      //
860059      return (0);
860060  }
```

kernel/fs/path_stat.c

«

Si veda la sezione i159.3.50.

```
870001  #include <kernel/fs.h>
870002  #include <errno.h>
870003  #include <kernel/proc.h>
870004  //-----------------------------------------------------------------------
870005  int
870006  path_stat (pid_t pid, const char *path, struct stat *buffer)
870007  {
870008      proc_t   *ps;
870009      inode_t  *inode;
870010      //
870011      // Get process.
870012      //
870013      ps = proc_reference (pid);
870014      //
870015      // Try to load the file inode.
870016      //
870017      inode = path_inode (pid, path);
870018      if (inode == NULL)
870019        {
870020          //
870021          // Cannot access the file: it does not exists or permissions are
870022          // not sufficient. Variable 'errno' is set by function
870023          // 'path_inode()'.
870024          //
870025          errset (errno);
870026          return (-1);
870027        }
870028      //
870029      // Inode loaded: update the buffer.
870030      //
870031      buffer->st_dev      = inode->sb->device;
870032      buffer->st_ino      = inode->ino;
870033      buffer->st_mode     = inode->mode;
870034      buffer->st_nlink    = inode->links;
870035      buffer->st_uid      = inode->uid;
870036      buffer->st_gid      = inode->gid;
870037      if (S_ISBLK (buffer->st_mode) || S_ISCHR (buffer->st_mode))
870038        {
870039          buffer->st_rdev = inode->direct[0];
870040        }
870041      else
870042        {
870043          buffer->st_rdev = 0;
870044        }
870045      buffer->st_size     = inode->size;
870046      buffer->st_atime    = inode->time;  // All times are the same for
870047      buffer->st_mtime    = inode->time;  // Minix 1 file system.
870048      buffer->st_ctime    = inode->time;  //
870049      buffer->st_blksize  = inode->sb->blksize;
870050      buffer->st_blocks   = inode->blkcnt;
870051      //
870052      // If the inode is a device special file, the 'st_rdev' value is
870053      // taken from the first direct zone (as of Minix 1 organization).
870054      //
870055      if (S_ISBLK(inode->mode) || S_ISCHR(inode->mode))
870056        {
870057          buffer->st_rdev = inode->direct[0];
870058        }
870059      else
870060        {
870061          buffer->st_rdev = 0;
870062        }
870063      //
870064      // Release the inode and return.
870065      //
870066      inode_put (inode);
870067      //
870068      // Return.
870069      //
870070      return (0);
870071  }
```

kernel/fs/path_umount.c

«

Si veda la sezione i159.3.41.

```
880001  #include <kernel/fs.h>
880002  #include <errno.h>
880003  #include <kernel/proc.h>
880004  //-----------------------------------------------------------------------
880005  int
880006  path_umount (pid_t pid, const char *path_mnt)
880007  {
```

```
880008      proc_t   *ps;
880009      dev_t     device;               // Device to mount.
880010      inode_t  *inode_mount_point;    // Original mount point.
880011      inode_t  *inode;                // Inode table.
880012      int       i;                    // Inode table index.
880013      //
880014      // Get process.
880015      //
880016      ps = proc_reference (pid);
880017      //
880018      // Verify to be the super user.
880019      //
880020      if (ps->euid != 0)
880021        {
880022          errset (EPERM);         // Operation not permitted.
880023          return (-1);
880024        }
880025      //
880026      // Get the directory mount point.
880027      //
880028      inode_mount_point = path_inode (pid, path_mnt);
880029      if (inode_mount_point == NULL)
880030        {
880031          errset (ENOENT);                // No such file or directory.
880032          return (-1);
880033        }
880034      //
880035      // Verify that the path is a directory.
880036      //
880037      if (!S_ISDIR (inode_mount_point->mode))
880038        {
880039          inode_put (inode_mount_point);
880040          errset (ENOTDIR);               // Not a directory.
880041          return (-1);
880042        }
880043      //
880044      // Verify that there is something attached.
880045      //
880046      device = inode_mount_point->sb_attached->device;
880047      if (device == 0)
880048        {
880049          //
880050          // There is nothing to unmount.
880051          //
880052          inode_put (inode_mount_point);
880053          errset (E_NOT_MOUNTED);         // Not mounted.
880054          return (-1);
880055        }
880056      //
880057      // Are there exactly two internal references? Let's explain:
880058      // the directory that act as mount point, should have one reference
880059      // because it is mounting something and another because it was just
880060      // opened again, a few lines above. If there are more references
880061      // it is wrong; if there are less, it is also wrong at this point.
880062      //
880063      if (inode_mount_point->references != 2)
880064        {
880065          inode_put (inode_mount_point);
880066          errset (EUNKNOWN);              // Unknown error.
880067          return (-1);
880068        }
880069      //
880070      // All data is available: find if there are open file inside
880071      // the file system to unmount. But first load the inode table
880072      // pointer.
880073      //
880074      inode = inode_reference ((dev_t) 0, (ino_t) 0);
880075      if (inode == NULL)
880076        {
880077          //
880078          // This error should not happen.
880079          //
880080          inode_put (inode_mount_point);
880081          errset (EUNKNOWN);              // Unknown error.
880082          return (-1);
880083        }
880084      //
880085      // Scan the inode table.
880086      //
880087      for (i = 0; i < INODE_MAX_SLOTS; i++)
880088        {
880089          if (inode[i].sb == inode_mount_point->sb_attached  &&
880090              inode[i].references > 0)
880091            {
880092              //
880093              // At least one file is open inside the super block to
880094              // release: cannot unmount.
880095              //
880096              inode_put (inode_mount_point);
880097              errset (EBUSY);             // Device or resource busy.
880098              return (-1);
880099            }
880100        }
880101      //
880102      // Can unmount: save and remove the super block memory;
880103      // clear the mount point reference and put inode.
880104      //
880105      inode_mount_point->sb_attached->changed          = 1;
880106      sb_save (inode_mount_point->sb_attached);
880107      //
880108      inode_mount_point->sb_attached->device           = 0;
```



```
880109      inode_mount_point->sb_attached->inode_mounted_on = NULL;
880110      inode_mount_point->sb_attached->blksize          = 0;
880111      inode_mount_point->sb_attached->options          = 0;
880112      //
880113      inode_mount_point->sb_attached                   = NULL;
880114      inode_mount_point->references                    = 0;
880115      inode_put (inode_mount_point);
880116      //
880117      inode_put (inode_mount_point);
880118      //
880119      return (0);
880120  }
```

## kernel/fs/path_unlink.c

«

Si veda la sezione i159.3.44.

```
890001  #include <kernel/fs.h>
890002  #include <errno.h>
890003  #include <kernel/proc.h>
890004  #include <libgen.h>
890005  #include <kernel/k_libc.h>
890006  //------------------------------------------------------------------------
890007  int
890008  path_unlink (pid_t pid, const char *path)
890009  {
890010      proc_t      *ps;
890011      inode_t     *inode_unlink;
890012      inode_t     *inode_directory;
890013      char         path_unlink[PATH_MAX];
890014      char         path_copy[PATH_MAX];
890015      char        *path_directory;
890016      char        *name_unlink;
890017      dev_t        device;
890018      off_t        start;
890019      char         buffer[SB_MAX_ZONE_SIZE];
890020      directory_t *dir = (directory_t *) buffer;
890021      int          status;
890022      ssize_t      size_read;
890023      ssize_t      size_written;
890024      int          d;                     // Directory buffer index.
890025      //
890026      // Get process.
890027      //
890028      ps = proc_reference (pid);
890029      //
890030      // Get full paths.
890031      //
890032      path_full (path, ps->path_cwd, path_unlink);
890033      strncpy (path_copy, path_unlink, PATH_MAX);
890034      path_directory = dirname (path_copy);
890035      //
890036      // Get the inode to be unlinked.
890037      //
890038      inode_unlink = path_inode (pid, path_unlink);
890039      if (inode_unlink == NULL)
890040        {
890041          return (-1);
890042        }
890043      //
890044      // If it is a directory, verify that it is empty.
890045      //
890046      if (S_ISDIR (inode_unlink->mode))
890047        {
890048          if (!inode_dir_empty (inode_unlink))
890049            {
890050              inode_put (inode_unlink);
890051              errset (ENOTEMPTY);         // Directory not empty.
890052              return (-1);
890053            }
890054        }
890055      //
890056      // Get the inode of the directory containing it.
890057      //
890058      inode_directory = path_inode (pid, path_directory);
890059      if (inode_directory == NULL)
890060        {
890061          inode_put (inode_unlink);
890062          return (-1);
890063        }
890064      //
890065      // Check if something is mounted on the directory.
890066      //
890067      if (inode_directory->sb_attached != NULL)
890068        {
890069          //
890070          // Must select the right directory.
890071          //
890072          device = inode_directory->sb_attached->device;
890073          inode_put (inode_directory);
890074          inode_directory = inode_get (device, 1);
890075          if (inode_directory == NULL)
890076            {
890077              inode_put (inode_unlink);
890078              return (-1);
890079            }
890080        }
890081      //
890082      // Check if write is allowed for the file system.
890083      //
890084      if (inode_directory->sb->options & MOUNT_RO)
```

```
890085        {
890086          errset (EROFS);         // Read-only file system.
890087          return (-1);
890088        }
890089      //
890090      // Verify access permissions for the directory. The number "3" means
890091      // that the user must have access permission and write permission:
890092      // "-wx" == 2+1 == 3.
890093      //
890094      status = inode_check (inode_directory, S_IFDIR, 3, ps->uid);
890095      if (status != 0)
890096        {
890097          errset (EPERM);                 // Operation not permitted.
890098          inode_put (inode_unlink);
890099          inode_put (inode_directory);
890100          return (-1);
890101        }
890102      //
890103      // Get the base name to be unlinked: this will alter the
890104      // original path.
890105      //
890106      name_unlink = basename (path_unlink);
890107      //
890108      // Read the directory content and try to locate the item to unlink.
890109      //
890110      for (start  = 0;
890111           start <  inode_directory->size;
890112           start += inode_directory->sb->blksize)
890113        {
890114          size_read = inode_file_read (inode_directory, start, buffer,
890115                                       inode_directory->sb->blksize,
890116                                       NULL);
890117          if (size_read < sizeof (directory_t))
890118            {
890119              break;
890120            }
890121          //
890122          // Scan the directory portion just read, for the item to unlink.
890123          //
890124          dir = (directory_t *) buffer;
890125          //
890126          for (d = 0; d < size_read; d += (sizeof (directory_t)), dir++)
890127            {
890128              if (dir->ino != 0                                       &&
890129                  strncmp (dir->name, name_unlink, NAME_MAX) == 0)
890130                {
890131                  //
890132                  // Found the corresponding item: unlink the inode.
890133                  //
890134                  dir->ino     = 0;
890135                  //
890136                  // Update the directory inside the file system.
890137                  //
890138                  size_written = inode_file_write (inode_directory, start,
890139                                                   buffer, size_read);
890140                  if (size_written != size_read)
890141                    {
890142                      //
890143                      // Write problem: just tell.
890144                      //
890145                      k_printf ("kernel alert: directory write error!\n");
890146                    }
890147                  //
890148                  // Update directory inode and put inode. If the unlinked
890149                  // inode was a directory, the parent directory inode
890150                  // must reduce the file system link count.
890151                  //
890152                  if (S_ISDIR (inode_unlink->mode))
890153                    {
890154                      inode_directory->links--;
890155                    }
890156                  inode_directory->time = k_time (NULL);
890157                  inode_directory->changed = 1;
890158                  inode_put (inode_directory);
890159                  //
890160                  // Reduce link inside unlinked inode and put inode.
890161                  //
890162                  inode_unlink->links--;
890163                  inode_unlink->changed = 1;
890164                  inode_unlink->time = k_time (NULL);
890165                  inode_put (inode_unlink);
890166                  //
890167                  // Just return, as the work is done.
890168                  //
890169                  return (0);
890170                }
890171            }
890172        }
890173      //
890174      // At this point, it was not possible to unlink the file.
890175      //
890176      inode_put (inode_unlink);
890177      inode_put (inode_directory);
890178      errset (EUNKNOWN);          // Unknown error.
890179      return (-1);
890180  }
```



## kernel/fs/sb_inode_status.c

«

Si veda la sezione i159.3.45.

```
900001  #include <kernel/fs.h>
900002  #include <errno.h>
900003  //-------------------------------------------------------------------------
900004  int
900005  sb_inode_status (sb_t *sb, ino_t ino)
900006  {
900007      int map_element;
900008      int map_bit;
900009      int map_mask;
900010      //
900011      // Check arguments.
900012      //
900013      if (ino == 0 || sb == NULL)
900014        {
900015          errset (EINVAL);        // Invalid argument.
900016          return (-1);
900017        }
900018      //
900019      //  Calculate the map element, the map bit and the map mask.
900020      //
900021      map_element = ino / 16;
900022      map_bit     = ino % 16;
900023      map_mask    = 1 << map_bit;
900024      //
900025      // Check the inode and return.
900026      //
900027      if (sb->map_inode[map_element] & map_mask)
900028        {
900029          return (1);     // True.
900030        }
900031      else
900032        {
900033          return (0);     // False.
900034        }
900035  }
```

## kernel/fs/sb_mount.c

«

Si veda la sezione i159.3.46.

```
910001  #include <kernel/fs.h>
910002  #include <errno.h>
910003  #include <kernel/devices.h>
910004  //-------------------------------------------------------------------------
910005  sb_t *
910006  sb_mount (dev_t device, inode_t **inode_mnt, int options)
910007  {
910008      sb_t   *sb;
910009      ssize_t size_read;
910010      addr_t  start;
910011      int     m;
910012      size_t  size_sb;
910013      size_t  size_map;
910014      //
910015      // Find if it is already mounted.
910016      //
910017      sb = sb_reference (device);
910018      if (sb != NULL)
910019        {
910020          errset (EBUSY);         // Device or resource busy: device
910021          return (NULL);          // already mounted.
910022        }
910023      //
910024      // Find if '*inode_mnt' is already mounting something.
910025      //
910026      if (*inode_mnt != NULL && (*inode_mnt)->sb_attached != NULL)
910027        {
910028          errset (EBUSY);         // Device or resource busy: mount point
910029          return (NULL);          // already used.
910030        }
910031      //
910032      // The inode is not yet mounting anything, or it is new: find a free
910033      // slot inside the super block table.
910034      //
910035      sb = sb_reference ((dev_t) -1);
910036      if (sb == NULL)
910037        {
910038          errset (EBUSY);         // Device or resource busy:
910039          return (NULL);          // no free slots.
910040        }
910041      //
910042      // A free slot was found: the super block header must be loaded, but
910043      // before it is necessary to calculate the header size to be read.
910044      //
910045      size_sb = offsetof (sb_t, device);
910046      //
910047      // Then fix the starting point.
910048      //
910049      start  = 1024;                      // After boot block.
910050      //
910051      // Read the file system super block header.
910052      //
910053      size_read = dev_io ((pid_t) -1, device, DEV_READ, start, sb,
910054                          size_sb, NULL);
910055      if (size_read != size_sb)
910056        {
910057          errset (EIO);                   // I/O error.
```

```
910058          return (NULL);
910059        }
910060      //
910061      // Save some more data.
910062      //
910063      sb->device           =  device;
910064      sb->options          =  options;
910065      sb->inode_mounted_on = *inode_mnt;
910066      sb->blksize          =  (1024 << sb->log2_size_zone);
910067      //
910068      // Check if the super block data is valid.
910069      //
910070      if (sb->magic_number != 0x137F)
910071        {
910072          errset (ENODEV);        // No such device: unsupported
910073          sb->device = 0;         // file system type.
910074          return (NULL);
910075        }
910076      if (sb->map_inode_blocks > SB_MAX_INODE_BLOCKS)
910077        {
910078          errset (E_MAP_INODE_TOO_BIG);
910079          return (NULL);
910080        }
910081      if (sb->map_zone_blocks > SB_MAX_ZONE_BLOCKS)
910082        {
910083          errset (E_MAP_ZONE_TOO_BIG);
910084          return (NULL);
910085        }
910086      if (sb->blksize > SB_MAX_ZONE_SIZE)
910087        {
910088          errset (E_DATA_ZONE_TOO_BIG);
910089          return (NULL);
910090        }
910091      //
910092      // A right super block header was loaded from disk, now load the
910093      // super block inode bit map.
910094      //
910095      start  = 1024;                              // After boot block.
910096      start += 1024;                              // After super block.
910097      for (m = 0; m < SB_MAP_INODE_SIZE; m++)     //
910098        {                                         // Reset map in memory,
910099          sb->map_inode[m] = 0xFFFF;              // before loading.
910100        }                                         //
910101      size_map = sb->map_inode_blocks * 1024;
910102      size_read = dev_io ((pid_t) -1, sb->device, DEV_READ, start,
910103                          sb->map_inode, size_map, NULL);
910104      if (size_read != size_map)
910105        {
910106          errset (EIO);                   // I/O error.
910107          return (NULL);
910108        }
910109      //
910110      // Load the super block zone bit map.
910111      //
910112      start  = 1024;                              // After boot block.
910113      start += 1024;                              // After super block.
910114      start += (sb->map_inode_blocks * 1024);     // After inode bit map.
910115      for (m = 0; m < SB_MAP_ZONE_SIZE; m++)      //
910116        {                                         // Reset map in memory,
910117          sb->map_zone[m] = 0xFFFF;               // before loading.
910118        }                                         //
910119      size_map = sb->map_zone_blocks * 1024;
910120      size_read = dev_io ((pid_t) -1, sb->device, DEV_READ, start,
910121                          sb->map_zone, size_map, NULL);
910122      if (size_read != size_map)
910123        {
910124          errset (EIO);                   // I/O error.
910125          return (NULL);
910126        }
910127      //
910128      // Check the inode that should mount the super block. If
910129      // '*inode_mnt' is 'NULL', then it is meant to be the first mount of
910130      // the root file system. In such case, the inode must be loaded too,
910131      // and the value for '*inode_mnt' must be modified.
910132      //
910133      if (*inode_mnt == NULL)
910134        {
910135          *inode_mnt = inode_get (device, 1);
910136        }
910137      //
910138      // Check for a valid value.
910139      //
910140      if (*inode_mnt == NULL)
910141        {
910142          //
910143          // This is bad!
910144          //
910145          errset (EUNKNOWN);              // Unknown error.
910146          return (NULL);
910147        }
910148      //
910149      // A valid inode is available for the mount.
910150      //
910151      (*inode_mnt)->sb_attached  = sb;
910152      //
910153      // Return the super block pointer.
910154      //
910155      return (sb);
910156  }
```



## kernel/fs/sb_reference.c

«

Si veda la sezione i159.3.47.

```
920001  #include <kernel/fs.h>
920002  #include <errno.h>
920003  //------------------------------------------------------------------------
920004  sb_t *
920005  sb_reference (dev_t device)
920006  {
920007      int s;                              // Slot index.
920008      //
920009      // If device is zero, a reference to the whole table is returned.
920010      //
920011      if (device == 0)
920012        {
920013          return (sb_table);
920014        }
920015      //
920016      // If device is ((dev_t) -1), a reference to a free slot is
920017      // returned.
920018      //
920019      if (device == ((dev_t) -1))
920020        {
920021          for (s = 0; s < SB_MAX_SLOTS; s++)
920022            {
920023              if (sb_table[s].device == 0)
920024                {
920025                  return (&sb_table[s]);
920026                }
920027            }
920028          return (NULL);
920029        }
920030      //
920031      // A device was selected: find the super block associated to it.
920032      //
920033      for (s = 0; s < SB_MAX_SLOTS; s++)
920034        {
920035          if (sb_table[s].device == device)
920036            {
920037              return (&sb_table[s]);
920038            }
920039        }
920040      //
920041      // The super block was not found.
920042      //
920043      return (NULL);
920044  }
```

## kernel/fs/sb_save.c

«

Si veda la sezione i159.3.48.

```
930001  #include <kernel/fs.h>
930002  #include <errno.h>
930003  #include <kernel/devices.h>
930004  //------------------------------------------------------------------------
930005  int
930006  sb_save (sb_t *sb)
930007  {
930008      ssize_t size_written;
930009      addr_t  start;
930010      size_t  size_map;
930011      //
930012      // Check for valid argument.
930013      //
930014      if (sb == NULL)
930015        {
930016          errset (EINVAL);                        // Invalid argument.
930017          return (-1);
930018        }
930019      //
930020      // Check if the super block changed for some reason (only the
930021      // inode and the zone maps can change really).
930022      //
930023      if (!sb->changed)
930024        {
930025          //
930026          // Nothing to save.
930027          //
930028          return (0);
930029        }
930030      //
930031      // Something inside the super block changed: start the procedure to
930032      // save the inode map (recall that the super block header is not
930033      // saved, because it never changes).
930034      //
930035      start  = 1024;                              // After boot block.
930036      start += 1024;                              // After super block.
930037      size_map = sb->map_inode_blocks * 1024;
930038      size_written = dev_io ((pid_t) -1, sb->device, DEV_WRITE, start,
930039                  sb->map_inode, size_map, NULL);
930040      if (size_written != size_map)
930041        {
930042          //
930043          // Error writing the map.
930044          //
930045          errset (EIO);                   // I/O error.
930046          return (-1);
930047        }
930048      //
```

```
930049      // Start the procedure to save the zone map.
930050      //
930051      start  = 1024;                              // After boot block.
930052      start += 1024;                              // After super block.
930053      start += (sb->map_inode_blocks * 1024);     // After inode bit map.
930054      size_map = sb->map_zone_blocks * 1024;
930055      size_written = dev_io ((pid_t) -1, sb->device, DEV_WRITE, start,
930056                   sb->map_zone, size_map, NULL);
930057      if (size_written != size_map)
930058        {
930059          //
930060          // Error writing the map.
930061          //
930062          errset (EIO);                   // I/O error.
930063          return (-1);
930064        }
930065      //
930066      // Super block saved.
930067      //
930068      sb->changed = 0;
930069      //
930070      return (0);
930071  }
```

## kernel/fs/sb_table.c

«

Si veda la sezione i159.3.47.

```
940001  #include <kernel/fs.h>
940002  //-------------------------------------------------------------------------
940003  sb_t    sb_table[SB_MAX_SLOTS];
```

## kernel/fs/sb_zone_status.c

«

Si veda la sezione i159.3.45.

```
950001  #include <kernel/fs.h>
950002  #include <errno.h>
950003  //-------------------------------------------------------------------------
950004  int
950005  sb_zone_status (sb_t *sb, zno_t zone)
950006  {
950007      int map_element;
950008      int map_bit;
950009      int map_mask;
950010      //
950011      // Check arguments.
950012      //
950013      if (zone == 0 || sb == NULL)
950014        {
950015          errset (EINVAL);        // Invalid argument.
950016          return (-1);
950017        }
950018      //
950019      //  Calculate the map element, the map bit and the map mask.
950020      //
950021      map_element = zone / 16;
950022      map_bit     = zone % 16;
950023      map_mask    = 1 << map_bit;
950024      //
950025      // Check the zone and return.
950026      //
950027      if (sb->map_zone[map_element] & map_mask)
950028        {
950029          return (1);     // True.
950030        }
950031      else
950032        {
950033          return (0);     // False.
950034        }
950035  }
```

## kernel/fs/zone_alloc.c

«

Si veda la sezione i159.3.51.

```
960001  #include <kernel/fs.h>
960002  #include <kernel/devices.h>
960003  #include <errno.h>
960004  //-------------------------------------------------------------------------
960005  zno_t
960006  zone_alloc (sb_t *sb)
960007  {
960008      int   m;                    // Index inside the inode map.
960009      int   map_element;
960010      int   map_bit;
960011      int   map_mask;
960012      zno_t zone;
960013      char  buffer[SB_MAX_ZONE_SIZE];
960014      int   status;
960015      //
960016      // Verify if write is allowed.
960017      //
960018      if (sb->options & MOUNT_RO)
960019        {
960020          errset (EROFS);         // Read-only file system.
960021          return ((zno_t) 0);
960022        }
```



```
960023      //
960024      // Write allowed: scan the zone map, to find a free zone.
960025      // If a free zone can be found, allocate it inside the map.
960026      // Index 'm' starts from one, because the first bit of the
960027      // map is reserved for a 'zero' data-zone that does not
960028      // exist: the second bit is for the real first data-zone.
960029      //
960030      for (zone = 0, m = 1; m < (SB_MAP_ZONE_SIZE * 16); m++)
960031        {
960032          map_element = m / 16;
960033          map_bit     = m % 16;
960034          map_mask    = 1 << map_bit;
960035          if (!(sb->map_zone[map_element] & map_mask))
960036            {
960037              //
960038              // Found a free place: set the map.
960039              //
960040              sb->map_zone[map_element] |= map_mask;
960041              sb->changed = 1;
960042              //
960043              // The *second* bit inside the map is for the first data
960044              // zone (the zone after the inode table inside the file
960045              // system), because the first is for a special 'zero' data
960046              // zone, not really used.
960047              //
960048              zone = sb->first_data_zone + m - 1; // Found a free zone.
960049              //
960050              // If the zone is outside the disk size, let set the map
960051              // bit, but reset variable 'zone'.
960052              //
960053              if (zone >= sb->zones)
960054                {
960055                  zone = 0;
960056                }
960057              else
960058                {
960059                  break;
960060                }
960061            }
960062        }
960063      if (zone == 0)
960064        {
960065          errset (ENOSPC);        // No space left on device.
960066          return ((zno_t) 0);
960067        }
960068      //
960069      // A free zone was found and the map was modified inside
960070      // the super block in memory. The zone must be cleared.
960071      //
960072      status = zone_write (sb, zone, buffer);
960073      if (status != 0)
960074        {
960075          zone_free (sb, zone);
960076          return ((zno_t) 0);
960077        }
960078      //
960079      // A zone was allocated: return the number.
960080      //
960081      return (zone);
960082  }
```

## kernel/fs/zone_free.c

«

Si veda la sezione i159.3.51.

```
970001  #include <kernel/fs.h>
970002  #include <kernel/devices.h>
970003  #include <errno.h>
970004  //-------------------------------------------------------------------------
970005  int
970006  zone_free (sb_t *sb, zno_t zone)
970007  {
970008      int map_element;
970009      int map_bit;
970010      int map_mask;
970011      //
970012      // Check arguments.
970013      //
970014      if (sb == NULL || zone < sb->first_data_zone)
970015        {
970016          errset (EINVAL);        // Invalid argument.
970017          return (-1);
970018        }
970019      //
970020      // Calculate the map element, the map bit and the map mask.
970021      //
970022      // The *second* bit inside the map is for the first data-zone
970023      // (the zone after the inode table inside the file system),
970024      // because the first is for a special 'zero' data-zone, not
970025      // really used.
970026      //
970027      map_element = (zone - sb->first_data_zone + 1) / 16;
970028      map_bit     = (zone - sb->first_data_zone + 1) % 16;
970029      map_mask    = 1 << map_bit;
970030      //
970031      // Verify if the requested zone is inside the file system area.
970032      //
970033      if (zone >= sb->zones)
970034        {
970035          errset (EINVAL);        // Invalid argument.
970036          return (-1);
```

```
970037        }
970038      //
970039      // Free the zone and return.
970040      //
970041      if (sb->map_zone[map_element] & map_mask)
970042        {
970043          sb->map_zone[map_element] &= ~map_mask;
970044          sb->changed = 1;
970045          return (0);
970046        }
970047      else
970048        {
970049          errset (EUNKNOWN);              // The zone was already free.
970050          return (-1);
970051        }
970052  }
```

## kernel/fs/zone_read.c

«

Si veda la sezione i159.3.53.

```
980001  #include <sys/os16.h>
980002  #include <kernel/fs.h>
980003  #include <kernel/devices.h>
980004  #include <errno.h>
980005  //------------------------------------------------------------------------
980006  int
980007  zone_read (sb_t *sb, zno_t zone, void *buffer)
980008  {
980009      size_t   size_zone;
980010      off_t    off_start;
980011      ssize_t  size_read;
980012      //
980013      // Verify if the requested zone is inside the file system area.
980014      //
980015      if (zone >= sb->zones)
980016        {
980017          errset (EINVAL);        // Invalid argument.
980018          return (-1);
980019        }
980020      //
980021      // Calculate start position.
980022      //
980023      size_zone  = 1024 << sb->log2_size_zone;
980024      off_start  = zone;
980025      off_start *= size_zone;
980026      //
980027      // Read from device to the buffer.
980028      //
980029      size_read = dev_io ((pid_t) -1, sb->device, DEV_READ, off_start,
980030                          buffer, size_zone, NULL);
980031      if (size_read != size_zone)
980032        {
980033          errset (EIO);           // I/O error.
980034          return (-1);
980035        }
980036      else
980037        {
980038          return (0);
980039        }
980040  }
```

## kernel/fs/zone_write.c

«

Si veda la sezione i159.3.53.

```
990001  #include <kernel/fs.h>
990002  #include <kernel/devices.h>
990003  #include <errno.h>
990004  //------------------------------------------------------------------------
990005  int
990006  zone_write (sb_t *sb, zno_t zone, void *buffer)
990007  {
990008      size_t   size_zone;
990009      off_t    off_start;
990010      ssize_t  size_written;
990011      //
990012      // Verify if write is allowed.
990013      //
990014      if (sb->options & MOUNT_RO)
990015        {
990016          errset (EROFS);         // Read-only file system.
990017          return (-1);
990018        }
990019      //
990020      // Verify if the requested zone is inside the file system area.
990021      //
990022      if (zone >= sb->zones)
990023        {
990024          errset (EINVAL);        // Invalid argument.
990025          return (-1);
990026        }
990027      //
990028      // Write is allowed: calculate start position.
990029      //
990030      size_zone  = 1024 << sb->log2_size_zone;
990031      off_start  = zone;
990032      off_start *= size_zone;
990033      //
990034      // Write the buffer to the device.
```



```
990035      //
990036      size_written = dev_io ((pid_t) -1, sb->device, DEV_WRITE, off_start,
990037                             buffer, size_zone, NULL);
990038      if (size_written != size_zone)
990039        {
990040          errset (EIO);           // I/O error.
990041          return (-1);
990042        }
990043      else
990044        {
990045          return (0);
990046        }
990047  }
```

## os16: «kernel/ibm_i86.h»

«

Si veda la sezione u0.4.

```
1000001  #ifndef _KERNEL_IBM_I86_H
1000002  #define _KERNEL_IBM_I86_H 1
1000003
1000004  #include <stdint.h>
1000005  #include <size_t.h>
1000006  #include <kernel/memory.h>
1000007  #include <sys/types.h>
1000008  //------------------------------------------------------------------------
1000009  #define IBM_I86_VIDEO_MODE   0x02
1000010  #define IBM_I86_VIDEO_PAGES  4
1000011
1000012  #define IBM_I86_VIDEO_COLUMNS   80
1000013  #define IBM_I86_VIDEO_ROWS      25
1000014  #define IBM_I86_VIDEO_ADDRESS   0xB8000L, 0xB9000L, 0xBA000L, 0xBB000L
1000015  //------------------------------------------------------------------------
1000016  void     _int10_00 (uint16_t video_mode);
1000017  void     _int10_02 (uint16_t page, uint16_t position);
1000018  void     _int10_05 (uint16_t page);
1000019  uint16_t _int12    (void);
1000020  uint16_t _int13_00 (uint16_t drive);
1000021  uint16_t _int13_02 (uint16_t drive, uint16_t sectors,
1000022                      uint16_t cylinder, uint16_t head,
1000023                      uint16_t sector, void *buffer);
1000024  uint16_t _int13_03 (uint16_t drive, uint16_t sectors,
1000025                      uint16_t cylinder, uint16_t head,
1000026                      uint16_t sector, void *buffer);
1000027  uint16_t _int16_00 (void);
1000028  uint16_t _int16_01 (void);
1000029  uint16_t _int16_02 (void);
1000030
1000031  #define int10_00(video_mode)     (_int10_00 ((uint16_t) video_mode))
1000032  #define int10_02(page, position) (_int10_02 ((uint16_t) page, \
1000033                                               (uint16_t) position))
1000034  #define int10_05(page)           (_int10_05 ((uint16_t) page))
1000035  #define int12()                  ((unsigned int) _int12 ())
1000036
1000037  #define int13_00(drive)          ((unsigned int) \
1000038                                    _int13_00 ((uint16_t) drive))
1000039  #define int13_02(drive, sectors, cylinder, head, sector, buffer) \
1000040                                   ((unsigned int) \
1000041                                    _int13_02 ((uint16_t) drive, \
1000042                                               (uint16_t) sectors, \
1000043                                               (uint16_t) cylinder, \
1000044                                               (uint16_t) head, \
1000045                                               (uint16_t) sector, \
1000046                                               buffer))
1000047  #define int13_03(drive, sectors, cylinder, head, sector, buffer) \
1000048                                   ((unsigned int) \
1000049                                    _int13_03 ((uint16_t) drive, \
1000050                                               (uint16_t) sectors, \
1000051                                               (uint16_t) cylinder, \
1000052                                               (uint16_t) head, \
1000053                                               (uint16_t) sector, \
1000054                                               buffer))
1000055  #define int16_00()               ((unsigned int) _int16_00 ())
1000056  #define int16_01()               ((unsigned int) _int16_01 ())
1000057  #define int16_02()               ((unsigned int) _int16_02 ())
1000058  //------------------------------------------------------------------------
1000059  uint16_t _in_8   (uint16_t port);
1000060  uint16_t _in_16  (uint16_t port);
1000061  void     _out_8  (uint16_t port, uint16_t value);
1000062  void     _out_16 (uint16_t port, uint16_t value);
1000063
1000064  #define in_8(port)           ((unsigned int) _in_8 ((uint16_t) port))
1000065  #define in_16(port)          ((unsigned int) _in_16 ((uint16_t) port))
1000066  #define out_8(port, value)   (_out_8 ((uint16_t) port, \
1000067                                        (uint16_t) value))
1000068  #define out_16(port, value)  (_out_16 ((uint16_t) port, \
1000069                                         (uint16_t) value))
1000070  //------------------------------------------------------------------------
1000071  void _cli     (void);
1000072  void _sti     (void);
1000073
1000074  #define cli() (_cli ())
1000075  #define sti() (_sti ())
1000076  //------------------------------------------------------------------------
1000077  void irq_on  (unsigned int irq);
1000078  void irq_off (unsigned int irq);
1000079  //------------------------------------------------------------------------
1000080  void     _ram_copy      (segment_t org_seg, offset_t org_off,
1000081                           segment_t dst_seg, offset_t dst_off,
1000082                           uint16_t size);
1000083
```

```
#define ram_copy(org_seg, org_off, dst_seg, dst_off, size) \
                        (_ram_copy ((uint16_t) org_seg, \
                                    (uint16_t) org_off, \
                                    (uint16_t) dst_seg, \
                                    (uint16_t) dst_off, \
                                    (uint16_t) size))
//---------------------------------------------------------------------
void con_select     (int console);
void con_putc       (int console, int c);
void con_scroll     (int console);
int  con_char_wait  (void);
int  con_char_read  (void);
int  con_char_ready (void);
void con_init       (void);
//---------------------------------------------------------------------
#define DSK_MAX                 4
#define DSK_SECTOR_SIZE         512     // Fixed!

typedef struct {
    unsigned int  bios_drive;
    unsigned int  cylinders;
    unsigned int  heads;
    unsigned int  sectors;
    unsigned int  retry;
} dsk_t;

typedef struct {
    unsigned int  cylinder;
    unsigned int  head;
    unsigned int  sector;
} dsk_chs_t;
//---------------------------------------------------------------------
extern dsk_t dsk_table[DSK_MAX];
//---------------------------------------------------------------------
void   dsk_setup         (void);
int    dsk_reset         (int drive);
void   dsk_sector_to_chs (int drive, unsigned int sector,
                          dsk_chs_t *chs);
int    dsk_read_sectors  (int drive, unsigned int start_sector,
                          void *buffer, unsigned int n_sectors);
int    dsk_write_sectors (int drive, unsigned int start_sector,
                          void *buffer, unsigned int n_sectors);
size_t dsk_read_bytes    (int drive, off_t offset,
                          void *buffer, size_t count);
size_t dsk_write_bytes   (int drive, off_t offset,
                          void *buffer, size_t count);
//---------------------------------------------------------------------

#endif
```

kernel/ibm_i86/_cli.s

« 

Si veda la sezione u0.4.

```
.global __cli
;---------------------------------------------------------------------
.text
;---------------------------------------------------------------------
; Clear interrupt flag.
;---------------------------------------------------------------------
.align 2
__cli:
    cli
    ret
```

kernel/ibm_i86/_in_16.s

« 

Si veda la sezione u0.4.

```
.global __in_16
;---------------------------------------------------------------------
.text
;---------------------------------------------------------------------
; Port input word.
;---------------------------------------------------------------------
__in_16:
    enter #2, #0        ; 1 local variable.
    pushf
    cli
    pusha
    mov   dx, 4[bp]     ; 1st arg (port number).
    in    ax, dx
    mov   -2[bp], ax    ; Save AX.
    popa
    popf
    mov   ax, -2[bp]    ; AX is the function return value.
    leave
    ret
```

kernel/ibm_i86/_in_8.s

« 

Si veda la sezione u0.4.

```
.global __in_8
;---------------------------------------------------------------------
.text
;---------------------------------------------------------------------
; Port input byte.
;---------------------------------------------------------------------
__in_8:
    enter #2, #0        ; 1 local variable.
    pushf
    cli
    pusha
    mov   dx, 4[bp]     ; 1st arg (port number).
    in    al, dx
    mov   ah, #0
    mov   -2[bp], ax    ; Save AX.
    popa
    popf
    mov   ax, -2[bp]    ; AX is the function return value.
    leave
    ret
```

kernel/ibm_i86/_int10_00.s

« 

Si veda la sezione u0.4.

```
.global __int10_00
;---------------------------------------------------------------------
.text
;---------------------------------------------------------------------
; INT 0x10 - video - set video mode
;       AH = 0x00
;       AL = desired video mode:
;            0x00 = text 40x25  pages 8
;            0x01 = text 40x25  pages 8
;            0x02 = text 80x25  pages 4
;            0x03 = text 80x25  pages 4
;            0x07 = text 80x25  pages 4?
;
; Specify the display mode for the currently active display adapter.
;---------------------------------------------------------------------
.align 2
__int10_00:
    enter #0, #0        ; No local variables.
    pushf
    cli
    pusha
    mov   ah, #0x00
    mov   al, 4[bp]     ; 1st arg (video mode).
    int   #0x10
    popa
    popf
    leave
    ret
```

kernel/ibm_i86/_int10_02.s

« 

Si veda la sezione u0.4.

```
.global __int10_02
;---------------------------------------------------------------------
.text
;---------------------------------------------------------------------
; INT 0x10 - video - set cursor position
;       AH = 0x02
;       BH = page number:
;            0-7 in modes 0 and 1
;            0-3 in modes 2 and 3
;       DH = row (0x00 is top)
;       DL = column (0x00 is left)
;---------------------------------------------------------------------
.align 2
__int10_02:
    enter #0, #0        ; No local variables.
    pushf
    cli
    pusha
    mov   ah, #0x02
    mov   bh, #0x00
    mov   bh, 4[bp]     ; 1st arg (page).
    mov   dx, 6[bp]     ; 2nd arg (pos).
    int   #0x10
    popa
    popf
    leave
    ret
```

kernel/ibm_i86/_int10_05.s

« 

Si veda la sezione u0.4.

```
.global __int10_05
;---------------------------------------------------------------------
.text
;---------------------------------------------------------------------
; INT 0x10 - video - select active display page
;       AH = 0x05
;       AL = new page number (0x00 is the first)
;---------------------------------------------------------------------
.align 2
__int10_05:
    enter #0, #0        ; No local variables.
    pushf
    cli
```

```
1060014     pusha
1060015     mov   ah, #0x05
1060016     mov   bh, 4[bp]     ; 1st arg (page).
1060017     int   #0x10
1060018     popa
1060019     popf
1060020     leave
1060021     ret
```

kernel/ibm_i86/_int12.s

«

Si veda la sezione u0.4.

```
1070001 .global __int12
1070002 ;--------------------------------------------------------------------
1070003 .text
1070004 ;--------------------------------------------------------------------
1070005 ; INT 12 - bios - get memory size
1070006 ; Return:
1070007 ;       AX = kilobytes of contiguous memory starting at absolute address
1070008 ;            0x00000
1070009 ;
1070010 ; This call returns the contents of the word at absolute address
1070011 ; 0x00413.
1070012 ;--------------------------------------------------------------------
1070013 .align 2
1070014 __int12:
1070015     enter #2, #0        ; 1 local variable.
1070016     pushf
1070017     cli
1070018     pusha
1070019     int   #0x12
1070020     mov   -2[bp], ax    ; save AX.
1070021     popa
1070022     popf
1070023     mov   ax, -2[bp]    ; AX is the function return value.
1070024     leave
1070025     ret
```

kernel/ibm_i86/_int13_00.s

«

Si veda la sezione u0.4.

```
1080001 .global __int13_00
1080002 ;--------------------------------------------------------------------
1080003 .text
1080004 ;--------------------------------------------------------------------
1080005 ; INT 0x13 - disk - reset disk system
1080006 ;       AH = 0x00
1080007 ;       DL = drive (if bit 7 is set both hard disks and floppy disks
1080008 ;            reset)
1080009 ; Return:
1080010 ;       AH = status
1080011 ;       CF clear if successful (returned AH=0x00)
1080012 ;       CF set on error
1080013 ;--------------------------------------------------------------------
1080014 .align 2
1080015 __int13_00:
1080016     enter #2, #0        ; 1 local variable.
1080017     pushf
1080018     cli
1080019     pusha
1080020     mov   ah, #0x00
1080021     mov   dl, 4[bp]     ; 1st arg.
1080022     int   #0x13
1080023     mov   al, #0x00
1080024     mov   -2[bp], ax    ; save AX.
1080025     popa
1080026     popf
1080027     mov   ax, -2[bp]    ; AX is the function return value.
1080028     leave
1080029     ret
```

kernel/ibm_i86/_int13_02.s

«

Si veda la sezione u0.4.

```
1090001 .global __int13_02
1090002 ;--------------------------------------------------------------------
1090003 .text
1090004 ;--------------------------------------------------------------------
1090005 ; INT 0x13 - disk - read sectors into memory
1090006 ;       AH = 0x02
1090007 ;       AL = number of sectors to read (must be nonzero)
1090008 ;       CH = cylinder number (0-255)
1090009 ;       CL bit 6-7 =
1090010 ;            cylinder number (256-1023)
1090011 ;       CL bit 0-5 =
1090012 ;            sector number   (1-63)
1090013 ;       DH = head number     (0-255)
1090014 ;       DL = drive number (bit 7 set for hard disk)
1090015 ;       ES:BX -> data buffer
1090016 ; Return:
1090017 ;       CF set on error
1090018 ;       CF clear if successful
1090019 ;       AH = status (0x00 if successful)
1090020 ;       AL = number of sectors transferred (only valid if CF set for
1090021 ;            some BIOSes)
```



```
1090022 ;--------------------------------------------------------------------
1090023 .align 2
1090024 __int13_02:
1090025     enter #2, #0        ; 1 local variable.
1090026     pushf
1090027     cli
1090028     pusha
1090029     mov   ax, ds        ; Set ES the same as DS.
1090030     mov   es, ax        ;
1090031     mov   ax,  8[bp]    ; 3rd arg (cylinder). It must be splitted and
1090032     mov   ch, al        ; assigned to the right registers.
1090033     mov   cl, ah        ;
1090034     shl   cl, 1         ;
1090035     shl   cl, 1         ;
1090036     shl   cl, 1         ;
1090037     shl   cl, 1         ;
1090038     shl   cl, 1         ;
1090039     shl   cl, 1         ;
1090040     add   cl, 12[bp]    ; 5th arg (sector).
1090041     mov   dl,  4[bp]    ; 1st arg (drive).
1090042     mov   al,  6[bp]    ; 2nd arg (sectors to be read).
1090043     mov   dh, 10[bp]    ; 4th arg (head).
1090044     mov   bx, 14[bp]    ; 6th arg (buffer pointer).
1090045     mov   ah, #0x02
1090046     int   #0x13
1090047     mov   -2[bp], ax    ; save AX.
1090048     popa
1090049     popf
1090050     mov   ax, -2[bp]    ; AX is the function return value.
1090051     leave
1090052     ret
```

kernel/ibm_i86/_int13_03.s

«

Si veda la sezione u0.4.

```
1100001 .global __int13_03
1100002 ;--------------------------------------------------------------------
1100003 .text
1100004 ;--------------------------------------------------------------------
1100005 ; INT 0x13 - disk - write sectors to disk
1100006 ;       AH = 0x03
1100007 ;       AL = number of sectors to write (must be nonzero)
1100008 ;       CH = cylinder number (0-255)
1100009 ;       CL bit 6-7 =
1100010 ;            cylinder number (256-1023)
1100011 ;       CL bit 0-5 =
1100012 ;            sector number   (1-63)
1100013 ;       DH = head number     (0-255)
1100014 ;       DL = drive number (bit 7 set for hard disk)
1100015 ;       ES:BX -> data buffer
1100016 ; Return:
1100017 ;       CF set on error
1100018 ;       CF clear if successful
1100019 ;       AH = status (0x00 if successful)
1100020 ;       AL = number of sectors transferred (only valid if CF set for
1100021 ;            some BIOSes)
1100022 ;--------------------------------------------------------------------
1100023 .align 2
1100024 __int13_03:
1100025     enter #2, #0        ; 1 local variable.
1100026     pushf
1100027     cli
1100028     pusha
1100029     mov   ax, ds        ; Set ES the same as DS.
1100030     mov   es, ax        ;
1100031     mov   ax,  8[bp]    ; 3rd arg (cylinder). It must be splitted and
1100032     mov   ch, al        ; assigned to the right registers.
1100033     mov   cl, ah        ;
1100034     shl   cl, 1         ;
1100035     shl   cl, 1         ;
1100036     shl   cl, 1         ;
1100037     shl   cl, 1         ;
1100038     shl   cl, 1         ;
1100039     shl   cl, 1         ;
1100040     add   cl, 12[bp]    ; 5th arg (sector).
1100041     mov   dl,  4[bp]    ; 1st arg (drive).
1100042     mov   al,  6[bp]    ; 2nd arg (sectors to be written).
1100043     mov   dh, 10[bp]    ; 4th arg (head).
1100044     mov   bx, 14[bp]    ; 6th arg (buffer pointer).
1100045     mov   ah, #0x03
1100046     int   #0x13
1100047     mov   -2[bp], ax    ; save AX.
1100048     popa
1100049     popf
1100050     mov   ax, -2[bp]    ; AX is the function return value.
1100051     leave
1100052     ret
```

kernel/ibm_i86/_int16_00.s

«

Si veda la sezione u0.4.

```
1110001 .global __int16_00
1110002 ;--------------------------------------------------------------------
1110003 .text
1110004 ;--------------------------------------------------------------------
1110005 ; INT 0x16 - keyboard - get keystroke
1110006 ;       AH = 0x00
1110007 ; Return:
```

```
;       AH = BIOS scan code
;       AL = ASCII character
;--------------------------------------------------------------------------
.align 2
__int16_00:
    enter #2, #0        ; 1 local variable.
    pushf
    cli
    pusha
    mov   ah, #0x00
    int   #0x16
    mov   -2[bp], ax    ; Save AX.
    popa
    popf
    mov   ax, -2[bp]    ; AX is the function return value.
    leave
    ret
```

kernel/ibm_i86/_int16_01.s

«

Si veda la sezione u0.4.

```
.global __int16_01
;--------------------------------------------------------------------------
.text
;--------------------------------------------------------------------------
; INT 0x16 - keyboard - check for keystroke
;       AH = 0x01
; Return:
;       ZF set if no keystroke available
;       ZF clear if keystroke available
;           AH = BIOS scan code
;           AL = ASCII character
;
; If a keystroke is present, it is not removed from the keyboard buffer.
;--------------------------------------------------------------------------
.align 2
__int16_01:
    enter #2, #0        ; 1 local variable.
    pushf
    cli
    pusha
    mov   ah, #0x01
    int   #0x16
    jnz   __int16_01_ok
    mov   ax, #0        ; Put zero to AX, if no keystroke is available.
__int16_01_ok:
    mov   -2[bp], ax    ; Save AX.
    popa
    popf
    mov   ax, -2[bp]    ; AX is the function return value.
    leave
    ret
```

kernel/ibm_i86/_int16_02.s

«

Si veda la sezione u0.4.

```
.global __int16_02
;--------------------------------------------------------------------------
.text
;--------------------------------------------------------------------------
; INT 0x16 - keyboard - get shift flags
;       AH = 0x02
; Return:
;       AL = shift flags
;       AH might be destroyed
;
; bit 7    Insert active
; bit 6    CapsLock active
; bit 5    NumLock active
; bit 4    ScrollLock active
; bit 3    Alt key pressed
; bit 2    Ctrl key pressed
; bit 1    left shift key pressed
; bit 0    right shift key pressed
;--------------------------------------------------------------------------
.align 2
__int16_02:
    enter #2, #0        ; 1 local variable.
    pushf
    cli
    pusha
    mov   ah, #0x02
    int   #0x16
    mov   ah, #0        ; Reset AH.
    mov   -2[bp], ax    ; Save AX.
    popa
    popf
    mov   ax, -2[bp]    ; AX is the function return value.
    leave
    ret
```



kernel/ibm_i86/_out_16.s

«

Si veda la sezione u0.4.

```
.global __out_16
;--------------------------------------------------------------------------
.text
;--------------------------------------------------------------------------
; Port output word.
;--------------------------------------------------------------------------
.align 2
__out_16:
    enter #0, #0        ; No local variables.
    pushf
    cli
    pusha
    mov   dx, 4[bp]     ; 1st arg (port number).
    mov   ax, 6[bp]     ; 2nd arg (value).
    out   dx, ax
    popa
    popf
    leave
    ret
```

kernel/ibm_i86/_out_8.s

«

Si veda la sezione u0.4.

```
.global __out_8
;--------------------------------------------------------------------------
.text
;--------------------------------------------------------------------------
; Port output byte.
;--------------------------------------------------------------------------
.align 2
__out_8:
    enter #0, #0        ; No local variables.
    pushf
    cli
    pusha
    mov   dx, 4[bp]     ; 1st arg (port number).
    mov   ax, 6[bp]     ; 2nd arg (value).
    out   dx, al
    popa
    popf
    leave
    ret
```

kernel/ibm_i86/_ram_copy.s

«

Si veda la sezione u0.4.

```
.global __ram_copy
;--------------------------------------------------------------------------
.text
;--------------------------------------------------------------------------
; Copy some bytes between segments.
;--------------------------------------------------------------------------
.align 2
__ram_copy:
    enter #0, #0        ; No local variables.
    pushf
    cli
    pusha
    mov   ax,  4[bp]    ; 1st arg (source segment).
    mov   si,  6[bp]    ; 2nd arg (source offset).
    mov   bx,  8[bp]    ; 3rd arg (destination segment).
    mov   di, 10[bp]    ; 4th arg (destination offset).
    mov   cx, 12[bp]    ; 5th arg (size).
    mov   dx, ds        ; save the data segment.
    mov   ds, ax        ; set DS.
    mov   es, bx        ; set ES.
    rep
    movsb               ; Copy the array of bytes.
    mov   ds, dx        ; Restore the data segment.
    mov   es, dx        ; Restore or fix the extra segment.
    popa
    popf
    leave
    ret
```

kernel/ibm_i86/_sti.s

«

Si veda la sezione u0.4.

```
.global __sti
;--------------------------------------------------------------------------
.text
;--------------------------------------------------------------------------
; Set interrupt flag.
;--------------------------------------------------------------------------
.align 2
__sti:
    sti
    ret
```

## kernel/ibm_i86/con_char_read.c

«

Si veda la sezione u0.4.

```
1180001  #include <kernel/ibm_i86.h>
1180002  #include <kernel/k_libc.h>
1180003  #include <sys/os16.h>
1180004  #include <sys/types.h>
1180005  #include <stdint.h>
1180006  //-------------------------------------------------------------------------
1180007  int
1180008  con_char_read (void)
1180009  {
1180010      int c;
1180011      c = int16_01 ();
1180012      //
1180013      // Remove special keys that are not used: they have zero in the low
1180014      // 8 bits, and something in the upper 8 bits.
1180015      //
1180016      if ((c & 0xFF00) && !(c & 0x00FF))
1180017        {
1180018          int16_00 ();    // Remove from buffer and return zero:
1180019          return (0);     // no key.
1180020        }
1180021      //
1180022      // A common key was pressed: filter only che low 8 bits.
1180023      //
1180024      c = c & 0x00FF;
1180025      if (c == 0)
1180026        {
1180027          return (c);     // There is no key.
1180028        }
1180029      if (c == '\r')      // Convert 'CR' to 'LF'.
1180030        {
1180031          c = '\n';
1180032        }
1180033      int16_00 ();        // Remove the key from buffer and return.
1180034      return (c);
1180035  }
```

## kernel/ibm_i86/con_char_ready.c

«

Si veda la sezione u0.4.

```
1190001  #include <kernel/ibm_i86.h>
1190002  #include <kernel/k_libc.h>
1190003  #include <sys/os16.h>
1190004  #include <sys/types.h>
1190005  #include <stdint.h>
1190006  //-------------------------------------------------------------------------
1190007  int
1190008  con_char_ready (void)
1190009  {
1190010      int c;
1190011      c = int16_01 ();
1190012      //
1190013      // Remove special keys that are not used: they have zero in the low
1190014      // 8 bits, and something in the upper 8 bits.
1190015      //
1190016      if ((c & 0xFF00) && !(c & 0x00FF))
1190017        {
1190018          int16_00 ();    // Remove from buffer and return zero:
1190019          return (0);     // no key.
1190020        }
1190021      //
1190022      // A common key was pressed: filter only che low 8 bits.
1190023      //
1190024      c = c & 0x00FF;
1190025      return (c);
1190026  }
```

## kernel/ibm_i86/con_char_wait.c

«

Si veda la sezione u0.4.

```
1200001  #include <kernel/ibm_i86.h>
1200002  #include <kernel/k_libc.h>
1200003  #include <sys/os16.h>
1200004  #include <sys/types.h>
1200005  #include <stdint.h>
1200006  //-------------------------------------------------------------------------
1200007  int
1200008  con_char_wait (void)
1200009  {
1200010      int c;
1200011      c = int16_00 ();
1200012      c = c & 0x00FF;
1200013      if (c == '\r')
1200014        {
1200015          c = '\n';
1200016        }
1200017      return (c);
1200018  }
```



## kernel/ibm_i86/con_init.c

«

Si veda la sezione u0.4.

```
1210001  #include <kernel/ibm_i86.h>
1210002  #include <kernel/k_libc.h>
1210003  #include <sys/os16.h>
1210004  #include <sys/types.h>
1210005  #include <stdint.h>
1210006  //-------------------------------------------------------------------------
1210007  void
1210008  con_init (void)
1210009  {
1210010      int page;
1210011      //
1210012      int10_00 (IBM_I86_VIDEO_MODE);
1210013      int10_05 (0);
1210014      //
1210015      for (page = 0; page < IBM_I86_VIDEO_PAGES; page++)
1210016        {
1210017          con_putc (page, '\n');
1210018        }
1210019  }
```

## kernel/ibm_i86/con_putc.c

«

Si veda la sezione u0.4.

```
1220001  #include <kernel/ibm_i86.h>
1220002  #include <kernel/k_libc.h>
1220003  #include <sys/os16.h>
1220004  #include <sys/types.h>
1220005  #include <stdint.h>
1220006  //-------------------------------------------------------------------------
1220007  void
1220008  con_putc (int console, int c)
1220009  {
1220010      static int      cursor[IBM_I86_VIDEO_PAGES];
1220011      static addr_t   address[] = {IBM_I86_VIDEO_ADDRESS};
1220012             addr_t   address_destination;
1220013             size_t   size_screen;
1220014             size_t   size_row;
1220015             uint16_t cell;
1220016             uint16_t attribute = 0x0700;
1220017             int      cursor_row;
1220018             int      cursor_column;
1220019             int      cursor_combined;
1220020
1220021      if (console < 0 || console >= IBM_I86_VIDEO_PAGES)
1220022        {
1220023          //
1220024          // No such console.
1220025          //
1220026          return;
1220027        }
1220028      //
1220029      // Calculate sizes.
1220030      //
1220031      size_row    = IBM_I86_VIDEO_COLUMNS;
1220032      size_screen = size_row * IBM_I86_VIDEO_ROWS;
1220033      //
1220034      // See if it is a special character, or if the cursor position
1220035      // requires a scroll up.
1220036      //
1220037      if (c == '\n')
1220038        {
1220039          con_scroll (console);
1220040          cursor[console] = (size_screen - size_row);
1220041        }
1220042      else if (c == '\b')
1220043        {
1220044          cursor[console]--;
1220045          if (cursor[console] < 0)
1220046            {
1220047              cursor[console] = 0;
1220048            }
1220049        }
1220050      else if (cursor[console] == (size_screen -1))
1220051        {
1220052          //
1220053          // Scroll up.
1220054          //
1220055          con_scroll (console);
1220056          //
1220057          cursor[console] -= size_row;
1220058        }
1220059      //
1220060      // If it is not a control character, print it.
1220061      //
1220062      if (c != '\n' && c != '\b')
1220063        {
1220064          //
1220065          // Write the character.
1220066          //
1220067          address_destination  = address[console];
1220068          address_destination += (cursor[console] * 2);
1220069          cell = (attribute | (c & 0x00FF));
1220070          //
1220071          mem_write (address_destination, &cell, sizeof (uint16_t));
1220072          //
1220073          // and an extra space after it (to be able to show the cursor).
```

```
1220074         //
1220075         cell = (attribute | ' ');
1220076         address_destination  += 2;
1220077         mem_write (address_destination, &cell, sizeof (uint16_t));
1220078         //
1220079         //
1220080         //
1220081         cursor[console]++;
1220082       }
1220083     //
1220084     // Update the cursor position on screen.
1220085     //
1220086     cursor_row      = cursor[console] / size_row;
1220087     cursor_column   = cursor[console] % size_row;
1220088     cursor_combined = (cursor_row << 8) | cursor_column;
1220089     //
1220090     // Set cursor position.
1220091     //
1220092     int10_02 (console, cursor_combined);
1220093 }
```

kernel/ibm_i86/con_scroll.c

«

Si veda la sezione u0.4.

```
1230001 #include <kernel/ibm_i86.h>
1230002 #include <kernel/k_libc.h>
1230003 #include <sys/os16.h>
1230004 #include <sys/types.h>
1230005 #include <stdint.h>
1230006 //---------------------------------------------------------------------------
1230007 void
1230008 con_scroll (int console)
1230009 {
1230010     static addr_t   address[] = {IBM_I86_VIDEO_ADDRESS};
1230011            addr_t   address_source;
1230012            addr_t   address_destination;
1230013            size_t   size_screen;
1230014            size_t   size_row;
1230015     static uint16_t empty_line[IBM_I86_VIDEO_COLUMNS];
1230016     //
1230017     size_row    = IBM_I86_VIDEO_COLUMNS;
1230018     size_screen = size_row * IBM_I86_VIDEO_ROWS;
1230019     //
1230020     // Scroll up.
1230021     //
1230022     address_source      = address[console];
1230023     address_source     += size_row * 2;
1230024     address_destination = address[console];
1230025     //
1230026     mem_copy (address_source, address_destination,
1230027               (size_t) ((size_screen - size_row) * 2));
1230028     //
1230029     address_destination  = address[console];
1230030     address_destination += ((size_screen - size_row) * 2);
1230031     //
1230032     mem_write (address_destination, &empty_line,
1230033                (size_t) (size_row * 2));
1230034 }
```

kernel/ibm_i86/con_select.c

«

Si veda la sezione u0.4.

```
1240001 #include <kernel/ibm_i86.h>
1240002 #include <kernel/k_libc.h>
1240003 #include <sys/os16.h>
1240004 #include <sys/types.h>
1240005 #include <stdint.h>
1240006 //---------------------------------------------------------------------------
1240007 void
1240008 con_select (int console)
1240009 {
1240010     //
1240011     // Variable 'console' gose from zero to 'IBM_I86_VIDEO_PAGES - 1'.
1240012     //
1240013     if (console >= 0 && console < IBM_I86_VIDEO_PAGES)
1240014       {
1240015         int10_05 (console);
1240016       }
1240017 }
```

kernel/ibm_i86/dsk_read_bytes.c

«

Si veda la sezione u0.4.

```
1250001 #include <kernel/ibm_i86.h>
1250002 #include <kernel/k_libc.h>
1250003 #include <sys/os16.h>
1250004 #include <sys/types.h>
1250005 #include <stdint.h>
1250006 //---------------------------------------------------------------------------
1250007 size_t
1250008 dsk_read_bytes (int drive, off_t offset, void *buffer, size_t count)
1250009 {
1250010     unsigned char *data_buffer = (unsigned char *) buffer;
1250011     int            status;
1250012     unsigned int   sector;
```



```
1250013     unsigned char  sector_buffer[DSK_SECTOR_SIZE];
1250014     int            i;
1250015     int            j = 0;
1250016     size_t         k = 0;
1250017
1250018     sector = offset / DSK_SECTOR_SIZE;
1250019     i      = offset % DSK_SECTOR_SIZE;
1250020
1250021     status = dsk_read_sectors (drive, sector, sector_buffer, 1);
1250022
1250023     if (status != 0)
1250024       {
1250025         return ((size_t) 0);
1250026       }
1250027
1250028     while (count)
1250029       {
1250030         for (; i < DSK_SECTOR_SIZE && count > 0;
1250031                i++, j++, k++, count--, offset++)
1250032           {
1250033             data_buffer[j] = sector_buffer[i];
1250034           }
1250035         if (count)
1250036           {
1250037             sector = offset / DSK_SECTOR_SIZE;
1250038             i      = offset % DSK_SECTOR_SIZE;
1250039             status = dsk_read_sectors (drive, sector, sector_buffer, 1);
1250040             if (status != 0)
1250041               {
1250042                 return (k);
1250043               }
1250044           }
1250045       }
1250046     return (k);
1250047 }
```

kernel/ibm_i86/dsk_read_sectors.c

«

Si veda la sezione u0.4.

```
1260001 #include <kernel/ibm_i86.h>
1260002 #include <kernel/k_libc.h>
1260003 #include <sys/os16.h>
1260004 #include <sys/types.h>
1260005 #include <stdint.h>
1260006 //---------------------------------------------------------------------------
1260007 int
1260008 dsk_read_sectors (int drive, unsigned int start_sector, void *buffer,
1260009                   unsigned int n_sectors)
1260010 {
1260011     int          status;
1260012     unsigned int retry;
1260013     unsigned int remaining;
1260014     dsk_chs_t    chs;
1260015     dsk_sector_to_chs (drive, start_sector, &chs);
1260016     remaining = dsk_table[drive].sectors - chs.sector + 1;
1260017     if (remaining < n_sectors)
1260018       {
1260019         status = dsk_read_sectors (drive, start_sector, buffer,
1260020                                    remaining);
1260021         if (status == 0)
1260022           {
1260023             status = dsk_read_sectors (drive, start_sector + remaining,
1260024                                        buffer, n_sectors - remaining);
1260025           }
1260026         return (status);
1260027       }
1260028     else
1260029       {
1260030         for (retry = 0; retry < dsk_table[drive].retry; retry++)
1260031           {
1260032             status = int13_02 (dsk_table[drive].bios_drive, n_sectors,
1260033                                chs.cylinder, chs.head, chs.sector,
1260034                                buffer);
1260035             status = status & 0x00F0;
1260036             if (status == 0)
1260037               {
1260038                 break;
1260039               }
1260040             else
1260041               {
1260042                 dsk_reset (drive);
1260043               }
1260044           }
1260045       }
1260046     if (status == 0)
1260047       {
1260048         return (0);
1260049       }
1260050     else
1260051       {
1260052         return (-1);
1260053       }
1260054 }
```

kernel/ibm_i86/dsk_reset.c «

Si veda la sezione u0.4.

```
#include <kernel/ibm_i86.h>
#include <kernel/k_libc.h>
#include <sys/os16.h>
#include <sys/types.h>
#include <stdint.h>
//------------------------------------------------------------------------
int
dsk_reset (int drive)
{
    unsigned int status;
    status = int13_00 (dsk_table[drive].bios_drive);
    if (status == 0)
      {
        return (0);
      }
    else
      {
        return (-1);
      }
}
```

kernel/ibm_i86/dsk_sector_to_chs.c «

Si veda la sezione u0.4.

```
#include <kernel/ibm_i86.h>
#include <kernel/k_libc.h>
#include <sys/os16.h>
#include <sys/types.h>
#include <stdint.h>
//------------------------------------------------------------------------
void
dsk_sector_to_chs (int drive, unsigned int sector, dsk_chs_t *chs)
{
    unsigned int sectors_per_cylinder;
    sectors_per_cylinder = (dsk_table[drive].sectors
                            * dsk_table[drive].heads);
    chs->cylinder        = sector / sectors_per_cylinder;
    sector               = sector % sectors_per_cylinder;
    chs->head            = sector / dsk_table[drive].sectors;
    sector               = sector % dsk_table[drive].sectors;
    chs->sector          = sector + 1;
}
```

kernel/ibm_i86/dsk_setup.c «

Si veda la sezione u0.4.

```
#include <kernel/ibm_i86.h>
//------------------------------------------------------------------------
void
dsk_setup (void)
{
  dsk_reset (0);
  dsk_table[0].bios_drive = 0x00;  // A: 1440 Kibyte floppy disk.
  dsk_table[0].cylinders  = 80;
  dsk_table[0].heads      = 2;
  dsk_table[0].sectors    = 18;
  dsk_table[0].retry      = 3;
  dsk_reset (1);
  dsk_table[1].bios_drive = 0x01;  // B: 1440 Kibyte floppy disk.
  dsk_table[1].cylinders  = 80;
  dsk_table[1].heads      = 2;
  dsk_table[1].sectors    = 18;
  dsk_table[1].retry      = 3;
  dsk_reset (2);
  dsk_table[2].bios_drive = 0x80;  // C: like a 2880 Kibyte floppy disk.
  dsk_table[2].cylinders  = 80;
  dsk_table[2].heads      = 2;
  dsk_table[2].sectors    = 36;
  dsk_table[2].retry      = 3;
  dsk_reset (3);
  dsk_table[3].bios_drive = 0x81;  // D: like a 2880 Kibyte floppy disk.
  dsk_table[3].cylinders  = 80;
  dsk_table[3].heads      = 2;
  dsk_table[3].sectors    = 36;
  dsk_table[3].retry      = 3;
}
```

kernel/ibm_i86/dsk_table.c «

Si veda la sezione u0.4.

```
#include <kernel/ibm_i86.h>
//------------------------------------------------------------------------
dsk_t dsk_table[DSK_MAX];
```

kernel/ibm_i86/dsk_write_bytes.c «

Si veda la sezione u0.4.

```
#include <kernel/ibm_i86.h>
#include <kernel/k_libc.h>
#include <sys/os16.h>
```



```
#include <sys/types.h>
#include <stdint.h>
//------------------------------------------------------------------------
size_t
dsk_write_bytes (int drive, off_t offset, void *buffer, size_t count)
{
    unsigned char *data_buffer = (unsigned char *) buffer;
    int            status;
    unsigned int   sector;
    unsigned char  sector_buffer[DSK_SECTOR_SIZE];
    int            i;
    int            j = 0;
    size_t         k = 0;
    size_t         m = 0;

    sector = offset / DSK_SECTOR_SIZE;
    i      = offset % DSK_SECTOR_SIZE;
    status = dsk_read_sectors (drive, sector, sector_buffer, 1);

    if (status != 0)
      {
        return ((size_t) 0);
      }

    while (count)
      {
        m = k;
        for (; i < DSK_SECTOR_SIZE && count > 0;
               i++, j++, k++, count--, offset++)
          {
            sector_buffer[i] = data_buffer[j];
          }
        status = dsk_write_sectors (drive, sector, sector_buffer, 1);
        if (status != 0)
          {
            return (m);
          }
        if (count)
          {
            sector = offset / DSK_SECTOR_SIZE;
            i      = offset % DSK_SECTOR_SIZE;
            status = dsk_read_sectors (drive, sector, sector_buffer, 1);
            if (status != 0)
              {
                return (m);
              }
          }
      }
    return (k);
}
```

kernel/ibm_i86/dsk_write_sectors.c «

Si veda la sezione u0.4.

```
#include <kernel/ibm_i86.h>
#include <kernel/k_libc.h>
#include <sys/os16.h>
#include <sys/types.h>
#include <stdint.h>
//------------------------------------------------------------------------
int
dsk_write_sectors (int drive, unsigned int start_sector, void *buffer,
                   unsigned int n_sectors)
{
    int          status;
    unsigned int retry;
    unsigned int remaining;
    dsk_chs_t    chs;
    dsk_sector_to_chs (drive, start_sector, &chs);
    remaining = dsk_table[drive].sectors - chs.sector + 1;
    if (remaining < n_sectors)
      {
        status = dsk_write_sectors (drive, start_sector,
                                    buffer, remaining);
        if (status == 0)
          {
            status = dsk_write_sectors (drive,
                                        start_sector + remaining,
                                        buffer, n_sectors - remaining);
          }
        return (status);
      }
    else
      {
        for (retry = 0; retry < dsk_table[drive].retry; retry++)
          {
            status = int13_03 (dsk_table[drive].bios_drive, n_sectors,
                               chs.cylinder, chs.head, chs.sector,
                               buffer);
            status = status & 0x00F0;
            if (status == 0)
              {
                break;
              }
            else
              {
                dsk_reset (drive);
              }
```

```
1320045           }
1320046         }
1320047       if (status == 0)
1320048         {
1320049           return (0);
1320050         }
1320051       else
1320052         {
1320053           return (-1);
1320054         }
1320055 }
```

kernel/ibm_i86/irq_off.c

«

Si veda la sezione u0.4.

```
1330001 #include <kernel/ibm_i86.h>
1330002 #include <kernel/k_libc.h>
1330003 #include <sys/os16.h>
1330004 #include <sys/types.h>
1330005 #include <stdint.h>
1330006 //---------------------------------------------------------------------
1330007 void
1330008 irq_off (unsigned int irq)
1330009 {
1330010     unsigned int mask;
1330011     unsigned int status;
1330012     if (irq > 7)
1330013       {
1330014         return; // Only XT IRQs are handled.
1330015       }
1330016     else
1330017       {
1330018         mask   = (1 << irq);
1330019         status = in_8 (0x21);
1330020         status = status | mask;
1330021         out_8 (0x21, status);
1330022       }
1330023 }
```

kernel/ibm_i86/irq_on.c

«

Si veda la sezione u0.4.

```
1340001 #include <kernel/ibm_i86.h>
1340002 #include <kernel/k_libc.h>
1340003 #include <sys/os16.h>
1340004 #include <sys/types.h>
1340005 #include <stdint.h>
1340006 //---------------------------------------------------------------------
1340007 void
1340008 irq_on (unsigned int irq)
1340009 {
1340010     unsigned int mask;
1340011     unsigned int status;
1340012     if (irq > 7)
1340013       {
1340014         return; // Only XT IRQs are handled.
1340015       }
1340016     else
1340017       {
1340018         mask   = ~(1 << irq);
1340019         status = in_8 (0x21);
1340020         status = status & mask;
1340021         out_8 (0x21, status);
1340022       }
1340023 }
```

## os16: «kernel/k_libc.h»

«

Si veda la sezione u0.5.

```
1350001 #ifndef _KERNEL_K_LIBC_H
1350002 #define _KERNEL_K_LIBC_H       1
1350003
1350004 #include <const.h>
1350005 #include <restrict.h>
1350006 #include <size_t.h>
1350007 #include <clock_t.h>
1350008 #include <time_t.h>
1350009 #include <sys/types.h>
1350010 #include <stdarg.h>
1350011
1350012 //---------------------------------------------------------------------
1350013 void    k_exit     (int status);
1350014 //---------------------------------------------------------------------
1350015 clock_t k_clock    (void);
1350016 int     k_stime    (time_t *timer);
1350017 time_t  k_time     (time_t *timer);
1350018 //---------------------------------------------------------------------
1350019 int     k_puts     (const char *string);
1350020 int     k_printf   (const char *restrict format, ...);
1350021 int     k_vprintf  (const char *restrict format, va_list arg);
1350022 int     k_vsprintf (char *restrict string, const char *restrict format,
1350023                     va_list arg);
1350024 void    k_perror   (const char *s);
1350025
1350026 int     k_kill     (pid_t pid, int sig);
```



```
1350027 int     k_open     (const char *file, int oflags, ...);
1350028 void    k_close    (int fd);
1350029 ssize_t k_read     (int fd, void *buffer, size_t count);
1350030 //---------------------------------------------------------------------
1350031
1350032 #endif
```

kernel/k_libc/k_clock.c

«

Si veda la sezione u0.5.

```
1360001 #include <kernel/k_libc.h>
1360002 //---------------------------------------------------------------------
1360003 extern clock_t _clock_ticks;    // uint32_t
1360004 //---------------------------------------------------------------------
1360005 clock_t
1360006 k_clock (void)
1360007 {
1360008     return (_clock_ticks);
1360009 }
```

kernel/k_libc/k_close.c

«

Si veda la sezione u0.5.

```
1370001 #include <kernel/k_libc.h>
1370002 #include <kernel/fs.h>
1370003 //---------------------------------------------------------------------
1370004 void
1370005 k_close (int fdn)
1370006 {
1370007     fd_close ((pid_t) 0, fdn);
1370008     return;
1370009 }
```

kernel/k_libc/k_exit.s

«

Si veda la sezione u0.5.

```
1380001 .global _k_exit
1380002 ;---------------------------------------------------------------------
1380003 .text
1380004 ;---------------------------------------------------------------------
1380005 .align 2
1380006 _k_exit:
1380007 halt:
1380008     hlt
1380009     jmp halt
```

kernel/k_libc/k_kill.c

«

Si veda la sezione u0.5.

```
1390001 #include <kernel/k_libc.h>
1390002 #include <kernel/proc.h>
1390003 //---------------------------------------------------------------------
1390004 int
1390005 k_kill (pid_t pid, int sig)
1390006 {
1390007     return (proc_sys_kill ((pid_t) 0, pid, sig));
1390008 }
```

kernel/k_libc/k_open.c

«

Si veda la sezione u0.5.

```
1400001 #include <kernel/k_libc.h>
1400002 #include <kernel/fs.h>
1400003 #include <stdarg.h>
1400004 #include <string.h>
1400005 #include <errno.h>
1400006 //---------------------------------------------------------------------
1400007 int
1400008 k_open (const char *file, int oflags, ...)
1400009 {
1400010     mode_t  mode;
1400011     va_list ap;
1400012     //
1400013     va_start (ap, oflags);
1400014     mode = va_arg (ap, mode_t);
1400015     //
1400016     if (file == NULL || strlen (file) == 0)
1400017       {
1400018         errset (EINVAL);        // Invalid argument.
1400019         return (-1);
1400020       }
1400021     return (fd_open ((pid_t) 0, file, oflags, mode));
1400022 }
```

## kernel/k_libc/k_perror.c

«

Si veda la sezione u0.5.

```
#include <kernel/k_libc.h>
#include <errno.h>
//---------------------------------------------------------------------------
void
k_perror (const char *s)
{
    //
    // If errno is zero, there is nothing to show.
    //
    if (errno == 0)
      {
        return;
      }
    //
    // Show the string if there is one.
    //
    if (s != NULL && strlen (s) > 0)
      {
        k_printf ("%s: ", s);
      }
    //
    // Show the translated error.
    //
    if (errfn[0] != 0 && errln != 0)
      {
        k_printf ("[%s:%u:%i] %s\n",
                  errfn, errln, errno, strerror (errno));
      }
    else
      {
        k_printf ("[%i] %s\n", errno, strerror (errno));
      }
}
```

## kernel/k_libc/k_printf.c

«

Si veda la sezione u0.5.

```
#include <stdarg.h>
#include <kernel/k_libc.h>
//---------------------------------------------------------------------------
int
k_printf  (const char *restrict format, ...)
{
    va_list ap;
    va_start (ap, format);
    return k_vprintf (format, ap);
}
```

## kernel/k_libc/k_puts.c

«

Si veda la sezione u0.5.

```
#include <sys/os16.h>
#include <kernel/devices.h>
#include <kernel/k_libc.h>
#include <string.h>
//---------------------------------------------------------------------------
int
k_puts (const char *string)
{
    dev_io ((pid_t) 0, DEV_TTY, DEV_WRITE, (off_t) 0, string,
            strlen (string), NULL);
    dev_io ((pid_t) 0, DEV_TTY, DEV_WRITE, (off_t) 0, "\n", 1, NULL);
    return 1;
}
```

## kernel/k_libc/k_read.c

«

Si veda la sezione u0.5.

```
#include <kernel/k_libc.h>
#include <kernel/fs.h>
//---------------------------------------------------------------------------
ssize_t
k_read (int fdn, void *buffer, size_t count)
{
    int     eof;
    ssize_t size;
    //
    eof = 0;
    //
    while (1)
      {
        size += fd_read ((pid_t) 0, fdn, buffer, count, &eof);
        if (size != 0 || eof)
          {
            break;
          }
      }
    return (size);
}
```



## kernel/k_libc/k_stime.c

«

Si veda la sezione u0.5.

```
#include <kernel/k_libc.h>
//---------------------------------------------------------------------------
extern time_t  _clock_seconds;  // uint32_t
//---------------------------------------------------------------------------
int
k_stime (time_t *timer)
{
    _clock_seconds = *timer;
    return (0);
}
```

## kernel/k_libc/k_time.c

«

Si veda la sezione u0.5.

```
#include <kernel/k_libc.h>
#include <stddef.h>
//---------------------------------------------------------------------------
extern time_t  _clock_seconds;  // uint32_t
//---------------------------------------------------------------------------
time_t
k_time (time_t *timer)
{
    if (timer != NULL)
      {
        *timer = _clock_seconds;
      }
    return (_clock_seconds);
}
```

## kernel/k_libc/k_vprintf.c

«

Si veda la sezione u0.5.

```
#include <sys/os16.h>
#include <kernel/devices.h>
#include <stdarg.h>
#include <kernel/k_libc.h>
#include <string.h>
//---------------------------------------------------------------------------
int
k_vprintf (const char *restrict format, va_list arg)
{
    char  string[BUFSIZ];
    int   ret;
    string[0] = 0;
    ret = k_vsprintf (string, format, arg);
    dev_io ((pid_t) 0, DEV_CONSOLE, DEV_WRITE, (off_t) 0, string,
            strlen (string), NULL);
    return ret;
}
```

## kernel/k_libc/k_vsprintf.c

«

Si veda la sezione u0.5.

```
#include <stdarg.h>
#include <kernel/k_libc.h>
#include <stdio.h>
//---------------------------------------------------------------------------
int
k_vsprintf (char *restrict string, const char *restrict format,
          va_list arg)
{
    int   ret;
    ret = vsnprintf (string, BUFSIZ, format, arg);
    return ret;
}
```

## os16: «kernel/main.h»

«

Si veda la sezione u0.6.

```
#ifndef _KERNEL_MAIN_H
#define _KERNEL_MAIN_H  1

#include <sys/types.h>

void  menu (void);
pid_t run  (char *path, char *argv[], char *envp[]);
int   main (int argc, char *argv[], char *envp[]);

#endif
```

## kernel/main/build.h

«

Si veda la sezione u0.6.

```
#define BUILD_DATE "2010.07.26 16:33:58"
```

Si veda la sezione u0.2.

```
1510001  .extern _main
1510002  .global ___mkargv
1510003  ;------------------------------------------------------------------------
1510004  ; Please note that, at the beginning, all the segment registers are
1510005  ; the same: CS==DS==ES==SS. But the data segments (DS, ES, SS) are meant
1510006  ; to be separated from the code, and they starts at: CS + __segoff.
1510007  ; The label "__segoff" is replaced by the linker with a constant value
1510008  ; (inside the code) with the segment offset to add to CS; this way
1510009  ; it is possibile to find where DS and ES should start.
1510010  ;------------------------------------------------------------------------
1510011  ; The following statement says that the code will start at "startup"
1510012  ; label.
1510013  ;------------------------------------------------------------------------
1510014  entry startup
1510015  ;------------------------------------------------------------------------
1510016  .text
1510017  ;------------------------------------------------------------------------
1510018  startup:
1510019      ;
1510020      ; Jump after initial data.
1510021      ;
1510022      jmp startup_code
1510023      ;
1510024  filler:
1510025      ;
1510026      ; After four bytes, from the start, there is the
1510027      ; magic number and other data.
1510028      ;
1510029      .space (0x0004 - (filler - startup))
1510030  magic:
1510031      ;
1510032      ; Add to "/etc/magic" the following line:
1510033      ;
1510034      ; 4  quad  0x6B65726E6F733136  os16 kernel
1510035      ;
1510036      .data4 0x6F733136   ; os16
1510037      .data4 0x6B65726E   ; kern
1510038      ;
1510039  segoff:                ;
1510040      .data2 __segoff     ; These values, for a kernel image,
1510041  etext:                 ; are not used.
1510042      .data2 __etext      ;
1510043  edata:                 ;
1510044      .data2 __edata      ;
1510045  ebss:                  ;
1510046      .data2 __end        ;
1510047  stack_size:            ;
1510048      .data2 0x0000       ;
1510049      ;
1510050      ; At the next label, the work begins.
1510051      ;
1510052  .align 2
1510053  startup_code:
1510054      ;
1510055      ; Check where we are. If we are at segment 0x1000,
1510056      ; then move to 0x3000.
1510057      ;
1510058      mov cx, cs
1510059      xor cx, #0x1000
1510060      jcxz move_code_from_0x1000_to_0x3000
1510061      ;
1510062      ; Check where we are. If we are at segment 0x3000,
1510063      ; then move to 0x0050, preserving the IVT and the BDA.
1510064      ;
1510065      mov cx, cs
1510066      xor cx, #0x3000
1510067      jcxz move_code_from_0x3000_to_0x0050
1510068      ;
1510069      ; Check where we are. If we are at segment 0x1050,
1510070      ; then jump to the main code.
1510071      ;
1510072      mov cx, cs
1510073      xor cx, #0x1050
1510074      jcxz main_code
1510075      ;
1510076      ; Otherwise, just halt.
1510077      ;
1510078      hlt
1510079      jmp startup_code
1510080      ;
1510081  move_code_from_0x1000_to_0x3000:
1510082      ;
1510083      cld                 ; Clear direction flag.
1510084      mov   ax, #0x3000   ; Set ES as the destination segment.
1510085      mov   es, ax        ;
1510086      mov   ax, #0x1000   ; Set DS as the source segment.
1510087      mov   ds, ax        ;
1510088      ;
1510089      mov   cx, #0x8000   ; Move 32768 words = 65536 byte (64 Kibyte).
1510090      mov   si, #0x0000   ; DS:SI == Source pointer
1510091      mov   di, #0x0000   ; ES:DI == Destination pointer
1510092      rep
1510093      movsw               ; Copy the array of words
1510094      ;
1510095      mov   ax, #0x4000   ; Set ES as the destination segment.
1510096      mov   es, ax        ;
1510097      mov   ax, #0x2000   ; Set DS as the source segment.
1510098      mov   ds, ax        ;
```



```
1510099      ;
1510100      mov   cx, #0x8000   ; Move 32768 words = 65536 byte (64 Kibyte).
1510101      mov   si, #0x0000   ; DS:SI == Source pointer
1510102      mov   di, #0x0000   ; ES:DI == Destination pointer
1510103      rep
1510104      movsw               ; Copy the array of words
1510105      ;
1510106      jmp far #0x3000:#0x0000     ; Go to the new kernel copy.
1510107      ;
1510108  move_code_from_0x3000_to_0x0050:
1510109      cld                 ; Clear direction flag.
1510110      ;
1510111      ; Text (instructions) is moved at segment 0x1050 (address 0x10500).
1510112      ;
1510113      mov   ax, #0x1050   ; Set ES as the destination segment.
1510114      mov   es, ax        ;
1510115      mov   ax, #0x3000   ; Set DS as the source segment.
1510116      mov   ds, ax        ;
1510117      ;
1510118      mov   cx, #0x8000   ; Move 32768 words = 65536 byte (64 Kibyte).
1510119      mov   si, #0x0000   ; DS:SI == Source pointer
1510120      mov   di, #0x0000   ; ES:DI == Destination pointer
1510121      rep
1510122      movsw               ; Copy the array of words
1510123      ;
1510124      ; Data is moved at segment 0x0050 (address 0x00500), before the
1510125      ; text segment.
1510126      ;
1510127      mov   ax, #0x0050   ; Set ES as the destination segment.
1510128      mov   es, ax        ;
1510129      mov   ax, #0x3000   ; Calculate where is the data segment:
1510130      add   ax, #__segoff ; it is at 0x3000 + __segoff.
1510131      mov   ds, ax        ; Set DS as the source segment.
1510132      ;
1510133      mov   cx, #0x8000   ; Move 32768 words = 65536 byte (64 Kibyte).
1510134      mov   si, #0x0000   ; DS:SI == Source pointer
1510135      mov   di, #0x0000   ; ES:DI == Destination pointer
1510136      rep
1510137      movsw               ; Copy the array of words
1510138      ;
1510139      jmp far #0x1050:#0x0000 ; Go to the new kernel copy.
1510140      ;
1510141  ;------------------------------------------------------------------------
1510142  main_code:
1510143      ;
1510144      ; Fix data segments!
1510145      ;
1510146      mov  ax, #0x0050
1510147      mov  ds, ax
1510148      mov  ss, ax
1510149      mov  es, ax
1510150      ;
1510151      ; Fix SP at the kernel stack bottom: the effective stack pointer
1510152      ; value should be 0x10000, but only 0x0000 can be written. At the
1510153      ; first push SP reaches 0xFFFE.
1510154      ;
1510155      mov sp, #0x0000
1510156      ;
1510157      ; Reset flags.
1510158      ;
1510159      push #0
1510160      popf
1510161      cli
1510162      ;
1510163      ; Call C main function, after kernel relocation and segments set up.
1510164      ;
1510165      push #0     ; This zero means NULL (envp[][] == NULL)
1510166      push #0     ; This zero means NULL (argv[][] == NULL)
1510167      push #0     ; This other zero means no arguments.
1510168      call _main
1510169      add  sp, #2
1510170      add  sp, #2
1510171      add  sp, #2
1510172      ;
1510173  .align 2
1510174  halt:
1510175      ;
1510176      ; It will never come back from the _kmain() call, but just for extra
1510177      ; security, loop forever.
1510178      ;
1510179      hlt
1510180      jmp halt
1510181      ;
1510182  ;------------------------------------------------------------------------
1510183  .align 2
1510184  ___mkargv:      ; Symbol '___mkargv' is used by Bcc inside the function
1510185      ret         ; 'main()' and must be present for a successful
1510186                  ; compilation.
1510187  ;------------------------------------------------------------------------
1510188  .align 2
1510189  .data
1510190  ;
1510191  ;------------------------------------------------------------------------
1510192  .align 2
1510193  .bss
```

## kernel/main/main.c

«

Si veda la sezione u0.6.

```
1520001  #include <kernel/main.h>
1520002  #include <errno.h>
1520003  #include <fcntl.h>
1520004  #include <kernel/main/build.h>
1520005  #include <kernel/diag.h>
1520006  #include <kernel/fs.h>
1520007  #include <kernel/ibm_i86.h>
1520008  #include <kernel/k_libc.h>
1520009  #include <kernel/proc.h>
1520010  #include <libgen.h>
1520011  #include <stdlib.h>
1520012  #include <sys/os16.h>
1520013  #include <sys/stat.h>
1520014  #include <sys/types.h>
1520015  #include <unistd.h>
1520016  //------------------------------------------------------------------------
1520017  int
1520018  main (int argc, char *argv[], char *envp[])
1520019  {
1520020      unsigned int key;
1520021      pid_t        pid;
1520022      char        *exec_argv[2];
1520023      int          status;
1520024      int          exit;
1520025      //
1520026      // Reset video and select the initial console.
1520027      //
1520028      tty_init ();
1520029      //
1520030      // Show compilation date and time.
1520031      //
1520032      k_printf ("os16 build %s ram %i Kibyte\n", BUILD_DATE, int12 ());
1520033      //
1520034      // Set up disk management.
1520035      //
1520036      dsk_setup ();
1520037      //
1520038      // Clear heap for diagnosis.
1520039      //
1520040      heap_clear ();
1520041      //
1520042      // Set up process management. Process set up need the file system
1520043      // root directory already available.
1520044      //
1520045      proc_init ();
1520046      //
1520047      // The kernel will run interactively.
1520048      //
1520049      menu ();
1520050      //
1520051      for (exit = 0; exit == 0;)
1520052        {
1520053          //
1520054          // While in kernel code, timer interrupt don't start the
1520055          // scheduler. The kernel must leave control to the scheduler
1520056          // via a null system call.
1520057          //
1520058          sys (SYS_0, NULL, 0);
1520059          //
1520060          // Back to work: read the keyboard from the TTY device.
1520061          //
1520062          dev_io ((pid_t) 0, DEV_TTY, DEV_READ, 0L, &key, 1, NULL);
1520063          //
1520064          // Depending on the key, do something.
1520065          //
1520066          if (key == 0)
1520067            {
1520068              //
1520069              // No key is ready in the buffer keyboard.
1520070              //
1520071              continue;
1520072            }
1520073          else
1520074            {
1520075              //
1520076              // Move back the cursor, so that next print will overwrite
1520077              // it.
1520078              //
1520079              k_printf ("\b");
1520080            }
1520081          //
1520082          // A key was pressed: start to check what it was.
1520083          //
1520084          switch (key)
1520085            {
1520086              case 'h':
1520087                  menu ();
1520088                  break;
1520089              case '1':
1520090                  k_kill ((pid_t) 1, SIGKILL);                  // init
1520091                  break;
1520092              case '2':
1520093              case '3':
1520094              case '4':
1520095              case '5':
1520096              case '6':
1520097              case '7':
1520098              case '8':
1520099              case '9':
1520100                  k_kill ((pid_t) (key - '0'),  SIGTERM);        // others
1520101                  break;
1520102              case 'A':
1520103              case 'B':
1520104              case 'C':
1520105              case 'D':
1520106              case 'E':
1520107              case 'F':
1520108                  k_kill ((pid_t) (key - 'A' + 10),  SIGTERM);   // others
1520109                  break;
1520110              case 'a':
1520111                  run ("/bin/aaa", NULL, NULL);
1520112                  break;
1520113              case 'b':
1520114                  run ("/bin/bbb", NULL, NULL);
1520115                  break;
1520116              case 'c':
1520117                  run ("/bin/ccc", NULL, NULL);
1520118                  break;
1520119              case 'f':
1520120                  print_file_list ();
1520121                  break;
1520122              case 'm':
1520123                  status = path_mount ((uid_t) 0, "/dev/dsk1", "/usr",
1520124                                       MOUNT_DEFAULT);
1520125                  if (status < 0)
1520126                    {
1520127                      k_perror (NULL);
1520128                    }
1520129                  break;
1520130              case 'M':
1520131                  status = path_umount ((uid_t) 0, "/usr");
1520132                  if (status < 0)
1520133                    {
1520134                      k_perror (NULL);
1520135                    }
1520136                  break;
1520137              case 'n':
1520138                  print_inode_list ();
1520139                  break;
1520140              case 'N':
1520141                  print_inode_zones_list ();
1520142                  break;
1520143              case 'l':
1520144                  k_kill ((pid_t) 1, SIGCHLD);
1520145                  break;
1520146              case 'p':
1520147                  k_printf ("\n");
1520148                  print_proc_list ();
1520149                  print_segments ();
1520150                  k_printf (" ");
1520151                  print_kmem ();
1520152                  k_printf (" ");
1520153                  print_time ();
1520154                  k_printf ("\n");
1520155                  print_mb_map ();
1520156                  k_printf ("\n");
1520157                  break;
1520158              case 'x':
1520159                  exit = 1;
1520160                  break;
1520161              case 'q':
1520162                  k_printf ("System halted!\n");
1520163                  return (0);
1520164                  break;
1520165            }
1520166        }
1520167      //
1520168      // Load init.
1520169      //
1520170      exec_argv[0] = "/bin/init";
1520171      exec_argv[1] = NULL;
1520172      pid = run ("/bin/init", exec_argv, NULL);
1520173      //
1520174      // Just sleep.
1520175      //
1520176      while (1)
1520177        {
1520178          sys (SYS_0, NULL, 0);
1520179        }
1520180      //
1520181      k_printf ("System halted!\n");
1520182      return (0);
1520183  }
```



## kernel/main/menu.c

«

Si veda la sezione u0.6.

```
1530001  #include <kernel/main.h>
1530002  #include <kernel/k_libc.h>
1530003  //------------------------------------------------------------------------
1530004  void
1530005  menu (void)
1530006  {
1530007      k_printf (
1530008  "+----------------------------------------------------------------+\n"
1530009  "| [h]      show this menu                                        |\n"
1530010  "| [p]      process status and memory map                         |\n"
1530011  "| [1]..[9] kill process  1 to 9                                  |\n"
```

```
"| [A]..[F] kill process 10 to 15                                    |\n"
"| [l]      send SIGCHLD to process 1                                |\n"
"| [a]..[c] run programs '/bin/aaa' to '/bin/ccc' in parallel        |\n"
"| [f]      system file status                                       |\n"
"| [n], [N] list of active inodes                                    |\n"
"| [m], [M] mount/umount '/dev/dsk1' at '/usr/'                      |\n"
"| [x]      exit interaction with kernel and start '/bin/init'       |\n"
"| [q]      quit kernel                                              |\n"
"'-----------------------------------------------------------------'\n"
);

}
```

kernel/main/run.c

«

Si veda la sezione u0.6.

```
#include <kernel/main.h>
#include <kernel/proc.h>
#include <kernel/k_libc.h>
#include <unistd.h>
//---------------------------------------------------------------------
pid_t
run (char *path, char *argv[], char *envp[])
{
    pid_t pid;
    //
    pid = fork ();
    if (pid == -1)
      {
        k_perror (NULL);
      }
    else if (pid == 0)
      {
        execve (path, argv, envp);
        k_perror (NULL);
        _exit (0);
      }
    return (pid);
}

```

os16: «kernel/memory.h»

«

Si veda la sezione u0.7.

```
#ifndef _KERNEL_MEMORY_H
#define _KERNEL_MEMORY_H       1

#include <stdint.h>
#include <stddef.h>
#include <sys/types.h>
//---------------------------------------------------------------------
#define MEM_BLOCK_SIZE     256  // 0x0100
#define MEM_MAX_BLOCKS    2560  // 655360/256 = 0xA0000/0x0100 = 0x0A00

extern uint16_t mb_table[MEM_MAX_BLOCKS/16]; // Memory blocks map.
//---------------------------------------------------------------------
typedef unsigned long int addr_t;
typedef unsigned int      segment_t;
typedef unsigned int      offset_t;
//---------------------------------------------------------------------
typedef struct {
    addr_t    address;
    segment_t segment;
    size_t    size;
} memory_t;
//---------------------------------------------------------------------
addr_t    address       (segment_t segment, offset_t offset);
//---------------------------------------------------------------------
uint16_t *mb_reference  (void);
ssize_t   mb_alloc      (addr_t address, size_t size);
void      mb_free       (addr_t address, size_t size);
int       mb_alloc_size (size_t size, memory_t *allocated);
//---------------------------------------------------------------------
void      mem_copy      (addr_t orig, addr_t dest, size_t size);
size_t    mem_read      (addr_t start, void *buffer, size_t size);
size_t    mem_write     (addr_t start, void *buffer, size_t size);
//---------------------------------------------------------------------

#endif
```

kernel/memory/address.c

«

Si veda la sezione u0.7.

```
#include <kernel/memory.h>
//---------------------------------------------------------------------
addr_t
address (segment_t segment, offset_t offset)
{
    addr_t a;
    a  = segment;
    a *= 16;
    a += offset;
    return (a);
}

```



```

```

kernel/memory/mb_alloc.c

«

Si veda la sezione u0.7.

```
#include <kernel/memory.h>
#include <kernel/ibm_i86.h>
#include <sys/os16.h>
#include <kernel/k_libc.h>
//---------------------------------------------------------------------
static int mb_block_set1   (int block);
//---------------------------------------------------------------------
ssize_t
mb_alloc (addr_t address, size_t size)
{
    unsigned int bstart;
    unsigned int bsize;
    unsigned int bend;
    unsigned int i;
    ssize_t      allocated = 0;
    addr_t       block_address;

    if (size == 0)
      {
        //
        // Zero means the maximum size.
        //
        bsize  = 0x10000L / MEM_BLOCK_SIZE;
      }
    else
      {
        bsize  = size / MEM_BLOCK_SIZE;
      }

    bstart = address / MEM_BLOCK_SIZE;

    if (size % MEM_BLOCK_SIZE)
      {
        bend = bstart + bsize;
      }
    else
      {
        bend = bstart + bsize - 1;
      }

    for (i = bstart; i <= bend; i++)
      {
        if (mb_block_set1 (i))
          {
            allocated += MEM_BLOCK_SIZE;
          }
        else
          {
            block_address = i;
            block_address *= MEM_BLOCK_SIZE;
            k_printf ("Kernel alert: mem block %04x, at address ", i);
            k_printf ("%05lx, already allocated!\n", block_address);
            break;
          }
      }
    return (allocated);
}

//---------------------------------------------------------------------
static int
mb_block_set1 (int block)
{
    int i         = block / 16;
    int j         = block % 16;
    uint16_t mask = 0x8000 >> j;
    if (mb_table[i] & mask)
      {
        return (0);     // The block is already set to 1 inside the map!
      }
    else
      {
        mb_table[i] = mb_table[i] | mask;
        return (1);
      }
}
```

kernel/memory/mb_alloc_size.c

«

Si veda la sezione u0.7.

```
#include <kernel/memory.h>
#include <kernel/ibm_i86.h>
#include <sys/os16.h>
#include <errno.h>
//---------------------------------------------------------------------
static int mb_block_status (int block);
//---------------------------------------------------------------------
```

```
1580008 int
1580009 mb_alloc_size (size_t size, memory_t *allocated)
1580010 {
1580011     unsigned int bsize;
1580012     unsigned int i;
1580013     unsigned int j;
1580014     unsigned int found = 0;
1580015     addr_t       alloc_addr;
1580016     ssize_t      alloc_size;
1580017
1580018     if (size == 0)
1580019       {
1580020         //
1580021         // Zero means the maximum size.
1580022         //
1580023         bsize = 0x10000L / MEM_BLOCK_SIZE;
1580024       }
1580025     else if (size % MEM_BLOCK_SIZE)
1580026       {
1580027         bsize = size / MEM_BLOCK_SIZE + 1;
1580028       }
1580029     else
1580030       {
1580031         bsize = size / MEM_BLOCK_SIZE;
1580032       }
1580033
1580034     for (i = 0; i < (MEM_MAX_BLOCKS - bsize) && !found; i++)
1580035       {
1580036         for (j = 0; j < bsize; j++)
1580037           {
1580038             found = !mb_block_status (i+j);
1580039             if (!found)
1580040               {
1580041                 i += j;
1580042                 break;
1580043               }
1580044           }
1580045       }
1580046
1580047     if (found && (j == bsize))
1580048       {
1580049         alloc_addr  = i - 1;
1580050         alloc_addr *= MEM_BLOCK_SIZE;
1580051         alloc_size  = bsize * MEM_BLOCK_SIZE;
1580052         alloc_size  = mb_alloc (alloc_addr, (size_t) alloc_size);
1580053         if (alloc_size <= 0)
1580054           {
1580055             errset (ENOMEM);
1580056             return (-1);
1580057           }
1580058         else if (alloc_size < size)
1580059           {
1580060             mb_free (alloc_addr, (size_t) alloc_size);
1580061             errset (ENOMEM);
1580062             return (-1);
1580063           }
1580064         else
1580065           {
1580066             allocated->address = alloc_addr;
1580067             allocated->segment = alloc_addr / 16;
1580068             allocated->size    = (size_t) alloc_size;
1580069           }
1580070         return (0);
1580071       }
1580072     else
1580073      {
1580074         errset (ENOMEM);
1580075         return (-1);
1580076      }
1580077 }
1580078 //-------------------------------------------------------------------------
1580079 static int
1580080 mb_block_status (int block)
1580081 {
1580082     int i         = block / 16;
1580083     int j         = block % 16;
1580084     uint16_t mask = 0x8000 >> j;
1580085     return ((int) (mb_table[i] & mask));
1580086 }
```

kernel/memory/mb_free.c

«

Si veda la sezione u0.7.

```
1590001 #include <kernel/memory.h>
1590002 #include <kernel/ibm_i86.h>
1590003 #include <sys/os16.h>
1590004 #include <kernel/k_libc.h>
1590005 //-------------------------------------------------------------------------
1590006 static int mb_block_set0   (int block);
1590007 //-------------------------------------------------------------------------
1590008 void
1590009 mb_free (addr_t address, size_t size)
1590010 {
1590011     unsigned int bstart;
1590012     unsigned int bsize;
1590013     unsigned int bend;
1590014     unsigned int i;
1590015     addr_t       block_address;
1590016     if (size == 0)
1590017       {
```



```
1590018         //
1590019         // Zero means the maximum size.
1590020         //
1590021         bsize  = 0x10000L / MEM_BLOCK_SIZE;
1590022       }
1590023     else
1590024       {
1590025         bsize  = size / MEM_BLOCK_SIZE;
1590026       }
1590027
1590028     bstart = address / MEM_BLOCK_SIZE;
1590029
1590030     if (size % MEM_BLOCK_SIZE)
1590031       {
1590032         bend = bstart + bsize;
1590033       }
1590034     else
1590035       {
1590036         bend = bstart + bsize - 1;
1590037       }
1590038
1590039     for (i = bstart; i <= bend; i++)
1590040       {
1590041         if (mb_block_set0 (i))
1590042           {
1590043             ;
1590044           }
1590045         else
1590046           {
1590047             block_address = i;
1590048             block_address *= MEM_BLOCK_SIZE;
1590049             k_printf ("Kernel alert: mem block %04x, at address ", i);
1590050             k_printf ("%05lx, already released!\n", block_address);
1590051           }
1590052       }
1590053 }
1590054 //-------------------------------------------------------------------------
1590055 static int
1590056 mb_block_set0 (int block)
1590057 {
1590058     int i         = block / 16;
1590059     int j         = block % 16;
1590060     uint16_t mask = 0x8000 >> j;
1590061     if (mb_table[i] & mask)
1590062       {
1590063         mb_table[i] = mb_table[i] & ~mask;
1590064         return (1);
1590065       }
1590066     else
1590067       {
1590068         return (0);     // The block is already set to 0 inside the map!
1590069       }
1590070 }
```

kernel/memory/mb_reference.c

«

Si veda la sezione u0.7.

```
1600001 #include <stdint.h>
1600002 #include <kernel/memory.h>
1600003 //-------------------------------------------------------------------------
1600004 uint16_t *
1600005 mb_reference (void)
1600006 {
1600007     return mb_table;
1600008 }
1600009
```

kernel/memory/mb_table.c

«

Si veda la sezione u0.7.

```
1610001 #include <kernel/memory.h>
1610002 #include <stdint.h>
1610003 //-------------------------------------------------------------------------
1610004 uint16_t mb_table[MEM_MAX_BLOCKS/16]; // Memory blocks map.
1610005 //-------------------------------------------------------------------------
```

kernel/memory/mem_copy.c

«

Si veda la sezione u0.7.

```
1620001 #include <kernel/memory.h>
1620002 #include <kernel/ibm_i86.h>
1620003 #include <sys/os16.h>
1620004 //-------------------------------------------------------------------------
1620005 void
1620006 mem_copy (addr_t orig, addr_t dest, size_t size)
1620007 {
1620008     segment_t seg_orig = orig / 16;
1620009     offset_t  off_orig = orig % 16;
1620010     segment_t seg_dest = dest / 16;
1620011     offset_t  off_dest = dest % 16;
1620012     ram_copy (seg_orig, off_orig, seg_dest, off_dest, size);
1620013 }
```

kernel/memory/mem_read.c

«

Si veda la sezione u0.7.

```
1630001 #include <kernel/memory.h>
1630002 #include <kernel/ibm_i86.h>
1630003 #include <sys/os16.h>
1630004 //-----------------------------------------------------------------------
1630005 size_t
1630006 mem_read (addr_t start, void *buffer, size_t size)
1630007 {
1630008     unsigned int      segment = start / 16;
1630009     unsigned int      offset  = start % 16;
1630010     unsigned long int end;
1630011     end  = start;
1630012     end += size;
1630013     if (end > 0x000FFFFFL)
1630014       {
1630015         size = 0x000FFFFFL - start;
1630016       }
1630017     ram_copy (segment, offset, seg_d (), (unsigned int) buffer, size);
1630018     return (size);
1630019 }
```

kernel/memory/mem_write.c

«

Si veda la sezione u0.7.

```
1640001 #include <kernel/memory.h>
1640002 #include <kernel/ibm_i86.h>
1640003 #include <sys/os16.h>
1640004 //-----------------------------------------------------------------------
1640005 size_t
1640006 mem_write (addr_t start, void *buffer, size_t size)
1640007 {
1640008     unsigned int      segment = start / 16;
1640009     unsigned int      offset  = start % 16;
1640010     unsigned long int end;
1640011     end  = start;
1640012     end += size;
1640013     if (end > 0x000FFFFFL)
1640014       {
1640015         size = 0x000FFFFFL - start;
1640016       }
1640017     ram_copy (seg_d (), (unsigned int) buffer, segment, offset, size);
1640018     return (size);
1640019 }
```

os16: «kernel/proc.h»

«

Si veda la sezione u0.8.

```
1650001 #ifndef _KERNEL_PROC_H
1650002 #define _KERNEL_PROC_H  1
1650003
1650004 #include <kernel/devices.h>
1650005 #include <kernel/memory.h>
1650006 #include <kernel/fs.h>
1650007 #include <kernel/tty.h>
1650008 #include <sys/types.h>
1650009 #include <sys/stat.h>
1650010 #include <sys/os16.h>
1650011 #include <stddef.h>
1650012 #include <stdint.h>
1650013 #include <time.h>
1650014
1650015 //-----------------------------------------------------------------------
1650016 #define CLOCK_FREQUENCY_DIVISOR     65535 // [1]
1650017 //
1650018 // [1]
1650019 // Internal clock frequency is (3579545/3) Hz.
1650020 // This value is divided by 65535 (0xFFFF) giving 18.2 Hz.
1650021 // The divisor value, 65535, if fixed!
1650022 //
1650023 //-----------------------------------------------------------------------
1650024 #define PROC_EMPTY                      0
1650025 #define PROC_CREATED                    1
1650026 #define PROC_READY                      2
1650027 #define PROC_RUNNING                    3
1650028 #define PROC_SLEEPING                   4
1650029 #define PROC_ZOMBIE                     5
1650030 //-----------------------------------------------------------------------
1650031 #define MAGIC_OS16              0x6F733136L  // os16
1650032 #define MAGIC_OS16_APPL         0x6170706CL  // appl
1650033 #define MAGIC_OS16_KERN         0x6B65726EL  // kern
1650034 //-----------------------------------------------------------------------
1650035 #define PROCESS_MAX  16                 // Process slots.
1650036
1650037 typedef struct {
1650038     pid_t             ppid;             // Parent PID.
1650039     pid_t             pgrp;             // Process group ID.
1650040     uid_t             uid;              // Real user ID
1650041     uid_t             euid;             // Effective user ID.
1650042     uid_t             suid;             // Saved user ID.
1650043     dev_t             device_tty;       // Controlling terminal.
1650044     char              path_cwd[PATH_MAX];
1650045                                         // Working directory path.
1650046     inode_t          *inode_cwd;        // Working directory inode.
1650047     int               umask;            // File creation mask.
1650048     unsigned long int sig_status;       // Active signals.
1650049     unsigned long int sig_ignore;       // Signals to be ignored.
1650050     clock_t           usage;            // Clock ticks CPU time usage.
1650051     unsigned int      status;
1650052     int               wakeup_events;    // Wake up for something.
1650053     int               wakeup_signal;    // Signal waited.
1650054     unsigned int      wakeup_timer;     // Seconds to wait for.
1650055     addr_t            address_i;
1650056     segment_t         segment_i;
1650057     size_t            size_i;
1650058     addr_t            address_d;
1650059     segment_t         segment_d;
1650060     size_t            size_d;
1650061     uint16_t          sp;
1650062     int               ret;
1650063     char              name[PATH_MAX];
1650064     fd_t              fd[FOPEN_MAX];
1650065 } proc_t;
1650066
1650067 extern proc_t   proc_table[PROCESS_MAX];
1650068 //-----------------------------------------------------------------------
1650069 typedef struct {
1650070     uint32_t filler0;
1650071     uint32_t magic0;
1650072     uint32_t magic1;
1650073     uint16_t segoff;
1650074     uint16_t etext;
1650075     uint16_t edata;
1650076     uint16_t ebss;
1650077     uint16_t ssize;
1650078 } header_t;
1650079 //-----------------------------------------------------------------------
1650080 void         _ivt_load          (void);
1650081 #define      ivt_load()         (_ivt_load ())
1650082 void         proc_init          (void);
1650083 void         proc_scheduler     (uint16_t *sp, segment_t *segment);
1650084 void         sysroutine         (uint16_t *sp, segment_t *segment,
1650085                                  uint16_t syscallnr, uint16_t msg_off,
1650086                                  uint16_t msg_size);
1650087 proc_t      *proc_reference     (pid_t pid);
1650088 //-----------------------------------------------------------------------
1650089 int          proc_sys_exec      (uint16_t *sp, segment_t *segment_d,
1650090                                  pid_t pid, const char *path,
1650091                                  unsigned int argc, char *arg_data,
1650092                                  unsigned int envc, char *env_data);
1650093 void         proc_sys_exit      (pid_t pid, int status);
1650094 pid_t        proc_sys_fork      (pid_t ppid, uint16_t sp);
1650095 int          proc_sys_kill      (pid_t pid_killer, pid_t pid_target,
1650096                                  int sig);
1650097 int          proc_sys_seteuid   (pid_t pid, uid_t euid);
1650098 int          proc_sys_setuid    (pid_t pid, uid_t uid);
1650099 sighandler_t proc_sys_signal    (pid_t pid, int sig,
1650100                                  sighandler_t handler);
1650101 pid_t        proc_sys_wait      (pid_t pid, int *status);
1650102 //-----------------------------------------------------------------------
1650103 void         proc_dump_memory   (pid_t pid, addr_t address,
1650104                                  size_t size, char *name);
1650105 void         proc_available     (pid_t pid);
1650106 pid_t        proc_find          (segment_t segment_d);
1650107 void         proc_sch_signals   (void);
1650108 void         proc_sch_terminals (void);
1650109 void         proc_sch_timers    (void);
1650110 void         proc_sig_chld      (pid_t parent, int sig);
1650111 void         proc_sig_cont      (pid_t pid, int sig);
1650112 void         proc_sig_core      (pid_t pid, int sig);
1650113 int          proc_sig_ignore    (pid_t pid, int sig);
1650114 void         proc_sig_off       (pid_t pid, int sig);
1650115 void         proc_sig_on        (pid_t pid, int sig);
1650116 int          proc_sig_status    (pid_t pid, int sig);
1650117 void         proc_sig_stop      (pid_t pid, int sig);
1650118 void         proc_sig_term      (pid_t pid, int sig);
1650119
1650120 #endif
```

kernel/proc/_isr.s

«

Si veda la sezione i159.8.1.

```
1660001 .extern _proc_scheduler
1660002 .extern _sysroutine
1660003 .global __ksp
1660004 .global __clock_ticks
1660005 .global __clock_seconds
1660006 .global isr_1C
1660007 .global isr_80
1660008 ;-------------------------------------------------------------------------
1660009 ; The kernel code segment starts at 0x10500 (segment 0x1050).
1660010 ; The kernel data segments start at 0x00500 (segment 0x0050).
1660011 ; To switch to the kernel data segments, DS, ES and SS are set to
1660012 ; 0x0050. To identify the kernel context, the DS register is checked:
1660013 ; if it is equal to 0x0050, it is the kernel.
1660014 ;-------------------------------------------------------------------------
1660015 .data
1660016 ;-------------------------------------------------------------------------
1660017 .align 2
1660018 proc_ss_0:       .word 0x0000
1660019 proc_sp_0:       .word 0x0000
1660020 proc_ss_1:       .word 0x0000
1660021 proc_sp_1:       .word 0x0000
1660022 proc_syscallnr:  .word 0x0000
1660023 proc_msg_offset: .word 0x0000
1660024 proc_msg_size:   .word 0x0000
```

```
__ksp:           .word 0x0000
__clock_ticks:
ticks_lo:        .word 0x0000
ticks_hi:        .word 0x0000
__clock_seconds:
seconds_lo:      .word 0x0000
seconds_hi:      .word 0x0000
;--------------------------------------------------------------------------
.text
;--------------------------------------------------------------------------
; IRQ 0: "timer".
; IRQ 0 is associated to INT 8, and after the BIOS work is done,
; INT 1C is called. Standard INT 1C has nothing to do, but is
; useful to call extra work for the timer. As the original BIOS
; interrupts are used, the INT 1C is reprogrammed, keeping intact
; the Standard INT 8.
;--------------------------------------------------------------------------
.align 2
isr_1C:
    ;--------------------------------------------------------------
    ; Inside the process stack, the CPU already put:
    ;
    ;       [omissis]
    ;       push flags
    ;       push cs
    ;       push ip
    ;--------------------------------------------------------------
    ;
    ; Save into process stack:
    ;
    push   es   ; extra segment
    push   ds   ; data segment
    push   di   ; destination index
    push   si   ; source index
    push   bp   ; base pointer
    push   bx   ; BX
    push   dx   ; DX
    push   cx   ; CX
    push   ax   ; AX
    ;
    ; Set the data segments to the kernel data area,
    ; so that the following variables can be accessed.
    ;
    mov ax, #0x0050  ; DS and ES.
    mov ds, ax       ;
    mov es, ax       ;
    ;
    ; Increment time counters, to keep time.
    ;
    add ticks_lo, #1    ; Clock ticks counter.
    adc ticks_hi, #0    ;
    ;
    mov dx, ticks_hi    ;
    mov ax, ticks_lo    ; DX := ticks % 18
    mov cx, #18         ;
    div cx              ;
    mov ax, #0          ; If the ticks value can be divided by 18,
    cmp ax, dx          ; the seconds is incremented by 1.
    jnz L1              ;
    add seconds_lo, #1  ;
    adc seconds_hi, #0  ;
    ;
L1: ; Save process stack registers into kernel data segment.
    ;
    mov proc_ss_0, ss   ; Save process stack segment.
    mov proc_sp_0, sp   ; Save process stack pointer.
    ;
    ; Check if it is already in kernel mode.
    ;
    mov dx, proc_ss_0
    mov ax, #0x0050     ; Kernel data area.
    cmp dx, ax
    je L2
    ;
    ; If we are here, a user process was interrupted.
    ; Switch to the kernel stack.
    ;
    mov ax, #0x0050     ; Kernel data area.
    mov ss, ax
    mov sp, __ksp
    ;
    ; Call the scheduler.
    ;
    push #proc_ss_0     ; &proc_ss_0
    push #proc_sp_0     ; &proc_sp_0
    call _proc_scheduler
    add  sp, #2
    add  sp, #2
    ;
    ; Restore process stack registers from kernel data segment.
    ;
    mov ss, proc_ss_0   ; Restore process stack segment.
    mov sp, proc_sp_0   ; Restore process stack pointer.
    ;
L2: ; Restore from process stack:
    ;
    pop    ax
    pop    cx
    pop    dx
    pop    bx
    pop    bp
```



```
    pop    si
    pop    di
    pop    ds
    pop    es
    ;
    ; Return from interrupt: will restore CS:IP and FLAGS
    ; from process stack.
    ;
    iret
;--------------------------------------------------------------------------
; Syscall.
;--------------------------------------------------------------------------
.align 2
isr_80:
    ;--------------------------------------------------------------
    ; Inside the process stack, we already have:
    ;       push #message_size
    ;       push &message_structure     ; the relative address of it
    ;       push #syscall_number
    ;       push #back_address  ; made by a call to _sys function
    ;       push flags          ; made by int #0x80
    ;       push cs             ; made by int #0x80
    ;       push ip             ; made by int #0x80
    ;--------------------------------------------------------------
    ;
    ; Save into process stack:
    ;
    push   es   ; extra segment
    push   ds   ; data segment
    push   di   ; destination index
    push   si   ; source index
    push   bp   ; base pointer
    push   bx   ; BX
    push   dx   ; DX
    push   cx   ; CX
    push   ax   ; AX
    ;
    ; Set the data segments to the kernel data area,
    ; so that the following variables can be accessed.
    ;
    mov ax, #0x0050     ; DS and ES.
    mov ds, ax          ;
    mov es, ax          ;
    ;
    ; Save process stack registers into kernel data segment.
    ;
    mov proc_ss_1, ss   ; Save process stack segment.
    mov proc_sp_1, sp   ; Save process stack pointer.
    ;
    ; Save some more data, from the system call.
    ;
    mov bp, sp
    mov ax, +26[bp]
    mov proc_syscallnr,  ax
    mov ax, +28[bp]
    mov proc_msg_offset, ax
    mov ax, +30[bp]
    mov proc_msg_size,   ax
    ;
    ; Check if it is already the kernel stack.
    ;
    mov dx, ss
    mov ax, #0x0050     ; Kernel data area.
    cmp dx, ax
    jne L3
    ;
    ; It is already the kernel stack, so, the variable "_ksp" is
    ; aligned to current stack pointer. This way, the first syscall
    ; can work without having to set the "_ksp" variable to some
    ; reasonable value.
    ;
    mov __ksp, sp
    ;
L3: ; Switch to the kernel stack.
    ;
    mov ax, #0x0050     ; Kernel data area.
    mov ss, ax
    mov sp, __ksp
    ;
    ; Call the external hardware interrupt handler.
    ;
    push proc_msg_size
    push proc_msg_offset
    push proc_syscallnr
    push #proc_ss_1     ; &proc_ss_1
    push #proc_sp_1     ; &proc_sp_1
    call _sysroutine
    add  sp, #2
    add  sp, #2
    add  sp, #2
    add  sp, #2
    add  sp, #2
    ;
    ; Restore process stack registers from kernel data segment.
    ;
    mov ss, proc_ss_1   ; Restore process stack segment.
    mov sp, proc_sp_1   ; Restore process stack pointer.
    ;
    ; Restore from process stack:
    ;
    pop    ax
```

```
1660227     pop    cx
1660228     pop    dx
1660229     pop    bx
1660230     pop    bp
1660231     pop    si
1660232     pop    di
1660233     pop    ds
1660234     pop    es
1660235     ;
1660236     ; Return from interrupt: will restore CS:IP and FLAGS
1660237     ; from process stack.
1660238     ;
1660239     iret
```

## kernel/proc/_ivt_load.s

«

Si veda la sezione i159.8.2.

```
1670001 .extern isr_1C
1670002 .extern isr_80
1670003 .global __ivt_load
1670004 ;------------------------------------------------------------------------
1670005 .text
1670006 ;------------------------------------------------------------------------
1670007 ; Load IVT.
1670008 ;
1670009 ; Currently, only the timer function and the syscall are loaded.
1670010 ;------------------------------------------------------------------------
1670011 .align 2
1670012 __ivt_load:
1670013     enter #0, #0                ; No local variables.
1670014     pushf
1670015     cli
1670016     pusha
1670017     ;
1670018     mov   ax,   #0              ; Change the DS segment to 0.
1670019     mov   ds,   ax              ;
1670020     ;
1670021     mov   bx,   #112            ; Timer         INT 0x08 (8) --> 0x1C
1670022     mov   [bx], #isr_1C         ; offset
1670023     mov   bx,   #114            ;
1670024     mov   [bx], cs              ; segment
1670025     ;
1670026     mov   bx,   #512            ; Syscall       INT 0x80 (128)
1670027     mov   [bx], #isr_80         ; offset
1670028     mov   bx,   #514            ;
1670029     mov   [bx], cs              ; segment
1670030     ;
1670031     mov   ax,   #0x0050         ; Put the DS segment back to the right
1670032     mov   ds,   ax              ; value.
1670033     ;
1670034     ;
1670035     ;
1670036     popa
1670037     popf
1670038     leave
1670039     ret
```

## kernel/proc/proc_available.c

«

Si veda la sezione i159.8.3.

```
1680001 #include <kernel/proc.h>
1680002 //---------------------------------------------------------------------
1680003 void
1680004 proc_available (pid_t pid)
1680005 {
1680006     proc_table[pid].ppid          = -1;
1680007     proc_table[pid].pgrp          = -1;
1680008     proc_table[pid].uid           = -1;
1680009     proc_table[pid].euid          = -1;
1680010     proc_table[pid].suid          = -1;
1680011     proc_table[pid].sig_status    = 0;
1680012     proc_table[pid].sig_ignore    = 0;
1680013     proc_table[pid].usage         = 0;
1680014     proc_table[pid].status        = PROC_EMPTY;
1680015     proc_table[pid].wakeup_events = 0;
1680016     proc_table[pid].wakeup_signal = 0;
1680017     proc_table[pid].wakeup_timer  = 0;
1680018     proc_table[pid].segment_i     = -1;
1680019     proc_table[pid].address_i     = -1L;
1680020     proc_table[pid].size_i        = -1;
1680021     proc_table[pid].segment_d     = -1;
1680022     proc_table[pid].address_d     = -1L;
1680023     proc_table[pid].size_d        = -1;
1680024     proc_table[pid].sp            = 0;
1680025     proc_table[pid].ret           = 0;
1680026     proc_table[pid].inode_cwd     = 0;
1680027     proc_table[pid].path_cwd[0]   = 0;
1680028     proc_table[pid].umask         = 0;
1680029     proc_table[pid].name[0]       = 0;
1680030 }
```



## kernel/proc/proc_dump_memory.c

«

Si veda la sezione i159.8.4.

```
1690001 #include <kernel/proc.h>
1690002 #include <fcntl.h>
1690003 //---------------------------------------------------------------------
1690004 void
1690005 proc_dump_memory (pid_t pid, addr_t address, size_t size, char *name)
1690006 {
1690007     int     fdn;
1690008     char    buffer[SB_BLOCK_SIZE];
1690009     ssize_t size_written;
1690010     ssize_t size_written_total;
1690011     ssize_t size_read;
1690012     ssize_t size_read_total;
1690013     ssize_t size_total;
1690014     //
1690015     // Dump the code segment to disk.
1690016     //
1690017     fdn = fd_open (pid, name, (O_WRONLY|O_CREAT|O_TRUNC),
1690018                    (mode_t) (S_IFREG|00644));
1690019     if (fdn < 0)
1690020       {
1690021         //
1690022         // There is a problem: just let it go.
1690023         //
1690024         return;
1690025       }
1690026     //
1690027     // Fix size: (size_t) 0 is equivalent to (ssize_t) 0x10000.
1690028     //
1690029     size_total = size;
1690030     if (size_total == 0)
1690031       {
1690032         size_total = 0x10000;
1690033       }
1690034     //
1690035     // Read the memory and write it to disk.
1690036     //
1690037     for (size_read = 0, size_read_total = 0;
1690038          size_read_total < size_total;
1690039          size_read_total += size_read, address += size_read)
1690040       {
1690041         size_read = mem_read (address, buffer, SB_BLOCK_SIZE);
1690042         //
1690043         for (size_written = 0, size_written_total = 0;
1690044              size_written_total < size_read;
1690045              size_written_total += size_written)
1690046           {
1690047             size_written = fd_write (pid, fdn,
1690048                                      &buffer[size_written_total],
1690049                                      (size_t) (size_read - size_written_total));
1690050             //
1690051             if (size_written < 0)
1690052               {
1690053                 fd_close (pid, fdn);
1690054                 return;
1690055               }
1690056           }
1690057       }
1690058     fd_close (pid, fdn);
1690059 }
```

## kernel/proc/proc_find.c

«

Si veda la sezione i159.8.5.

```
1700001 #include <kernel/proc.h>
1700002 #include <kernel/k_libc.h>
1700003 #include <kernel/diag.h>
1700004 //---------------------------------------------------------------------
1700005 pid_t
1700006 proc_find (segment_t segment_d)
1700007 {
1700008     int    pid;
1700009     addr_t address_d;
1700010     for (pid = 0; pid < PROCESS_MAX; pid++)
1700011       {
1700012         if (proc_table[pid].segment_d == segment_d)
1700013           {
1700014             break;
1700015           }
1700016       }
1700017     if (pid >= PROCESS_MAX)
1700018       {
1700019         address_d  = segment_d;
1700020         address_d *= 16;
1700021         k_printf ("\n"
1700022                   "Kernel panic: cannot find the interrupted process "
1700023                   "inside the process table. "
1700024                   "The wanted process has data segment 0x%04x \n"
1700025                   "(effective address %05lx)!\n",
1700026                   (unsigned int) segment_d, address_d);
1700027         print_proc_list ();
1700028         print_segments ();
1700029         k_printf (" ");
1700030         print_kmem ();
1700031         k_printf (" ");
1700032         print_time ();
1700033         k_printf ("\n");
```

```
1700034         print_mb_map ();
1700035         k_printf ("\n");
1700036         k_exit (0);
1700037       }
1700038     return (pid);
1700039 }
```

## kernel/proc/proc_init.c

«

Si veda la sezione i159.8.6.

```
1710001 #include <kernel/proc.h>
1710002 #include <kernel/k_libc.h>
1710003 #include <string.h>
1710004 //---------------------------------------------------------------------
1710005 extern uint16_t _etext;
1710006 //---------------------------------------------------------------------
1710007 void
1710008 proc_init (void)
1710009 {
1710010     uint8_t  divisor_lo;
1710011     uint8_t  divisor_hi;
1710012     pid_t    pid;
1710013     int      fdn;                // File descriptor index;
1710014     addr_t   start;              // Used for effective memory addresses.
1710015     size_t   size;               // Used for memory allocation.
1710016     inode_t *inode;
1710017     sb_t    *sb;
1710018     //
1710019     // Clear interrupts (should already be cleared).
1710020     //
1710021     cli ();
1710022     //
1710023     // Load Interrupt vector table (IVT).
1710024     //
1710025     ivt_load ();
1710026     //
1710027     // Configure the clock: must be the original values, because
1710028     // the BIOS depends on it!
1710029     //
1710030     // Base frequency is 1193181 Hz and it should divided.
1710031     // Resulting frequency must be from 18.22 Hz and 1193181 Hz.
1710032     // The calculated value (the divisor) must be sent to the
1710033     // PIT (programmable interval timer), divided in two pieces.
1710034     //
1710035     divisor_lo = (CLOCK_FREQUENCY_DIVISOR & 0xFF);         // Low  byte.
1710036     divisor_hi = (CLOCK_FREQUENCY_DIVISOR / 0x100) & 0xFF; // High byte.
1710037     out_8 (0x43, 0x36);
1710038     out_8 (0x40, divisor_lo);        // Lower byte.
1710039     out_8 (0x40, divisor_hi);        // Higher byte.
1710040     //
1710041     // Set all memory reference to some invalid data.
1710042     //
1710043     for (pid = 0; pid < PROCESS_MAX; pid++)
1710044       {
1710045         proc_available (pid);
1710046       }
1710047     //
1710048     // Mount root file system.
1710049     //
1710050     inode = NULL;
1710051     sb = sb_mount (DEV_DSK0, &inode, MOUNT_DEFAULT);
1710052     if (sb == NULL || inode == NULL)
1710053       {
1710054         k_perror ("Kernel panic: cannot mount root file system:");
1710055         k_exit (0);
1710056       }
1710057     //
1710058     // Set up the process table with the kernel.
1710059     //
1710060     proc_table[0].ppid          = 0;
1710061     proc_table[0].pgrp          = 0;
1710062     proc_table[0].uid           = 0;
1710063     proc_table[0].euid          = 0;
1710064     proc_table[0].suid          = 0;
1710065     proc_table[0].device_tty    = DEV_UNDEFINED;
1710066     proc_table[0].sig_status    = 0;
1710067     proc_table[0].sig_ignore    = 0;
1710068     proc_table[0].usage         = 0;
1710069     proc_table[0].status        = PROC_RUNNING;
1710070     proc_table[0].wakeup_events = 0;
1710071     proc_table[0].wakeup_signal = 0;
1710072     proc_table[0].wakeup_timer  = 0;
1710073     proc_table[0].segment_i     = seg_i ();
1710074     proc_table[0].address_i     = seg_i ();
1710075     proc_table[0].address_i    *= 16;
1710076     proc_table[0].size_i        = (size_t) &_etext;
1710077     proc_table[0].segment_d     = seg_d ();
1710078     proc_table[0].address_d     = seg_d ();
1710079     proc_table[0].address_d    *= 16;
1710080     proc_table[0].size_d        = 0;     // Maximum size: 0x10000.
1710081     proc_table[0].sp            = 0;     // To be set at next interrupt.
1710082     proc_table[0].ret           = 0;
1710083     proc_table[0].umask         = 0022;  // Default umask.
1710084     proc_table[0].inode_cwd     = inode; // Root fs inode.
1710085     strncpy (proc_table[0].path_cwd, "/", PATH_MAX);
1710086     strncpy (proc_table[0].name, "os16 kernel", PATH_MAX);
1710087     //
1710088     // Ensure to have a terminated string.
1710089     //
1710090     proc_table[0].name[PATH_MAX-1] = 0;
```



```
1710091     //
1710092     // Reset file descriptors.
1710093     //
1710094     for (fdn = 0; fdn < OPEN_MAX; fdn++)
1710095       {
1710096         proc_table[0].fd[fdn].fl_flags = 0;
1710097         proc_table[0].fd[fdn].fd_flags = 0;
1710098         proc_table[0].fd[fdn].file     = NULL;
1710099       }
1710100     //
1710101     // Allocate memory for the code segment.
1710102     //
1710103     mb_alloc (proc_table[0].address_i, proc_table[0].size_i);
1710104     //
1710105     // Allocate memory for the data segment if different.
1710106     //
1710107     if (seg_d () != seg_i ())
1710108       {
1710109         mb_alloc (proc_table[0].address_d, proc_table[0].size_d);
1710110       }
1710111     //
1710112     // Allocate memory for the BIOS data area (BDA).
1710113     //
1710114     mb_alloc (0x00000L, 0x500);
1710115     //
1710116     // Allocate memory for the extra BIOS at the
1710117     // bottom of the 640 Kibyte.
1710118     //
1710119     start  = int12 ();
1710120     start *= 1024;
1710121     size   = 0xA0000L - start;
1710122     mb_alloc (start, size);
1710123     //
1710124     // Enable and disable hardware interrupts (IRQ).
1710125     //
1710126     irq_on  (0); // timer.
1710127     irq_on  (1); // enable keyboard
1710128     irq_off (2); //
1710129     irq_off (3); //
1710130     irq_off (4); //
1710131     irq_off (5); //
1710132     irq_on  (6); // floppy (must be on to let int 13 work)!
1710133     irq_off (7); //
1710134     //
1710135     // Interrupts activation.
1710136     //
1710137     sti ();
1710138 }
```

## kernel/proc/proc_reference.c

«

Si veda la sezione i159.8.7.

```
1720001 #include <kernel/proc.h>
1720002 //---------------------------------------------------------------------
1720003 proc_t *
1720004 proc_reference (pid_t pid)
1720005 {
1720006     if (pid >= 0 && pid < PROCESS_MAX)
1720007       {
1720008         return (&proc_table[pid]);
1720009       }
1720010     else
1720011       {
1720012         return (NULL);
1720013       }
1720014 }
```

## kernel/proc/proc_sch_signals.c

«

Si veda la sezione i159.8.8.

```
1730001 #include <kernel/proc.h>
1730002 //---------------------------------------------------------------------
1730003 void
1730004 proc_sch_signals (void)
1730005 {
1730006     pid_t  pid;
1730007     for (pid = 0; pid < PROCESS_MAX; pid++)
1730008       {
1730009         proc_sig_term (pid, SIGHUP);
1730010         proc_sig_term (pid, SIGINT);
1730011         proc_sig_core (pid, SIGQUIT);
1730012         proc_sig_core (pid, SIGILL);
1730013         proc_sig_core (pid, SIGABRT);
1730014         proc_sig_core (pid, SIGFPE);
1730015         proc_sig_term (pid, SIGKILL);
1730016         proc_sig_core (pid, SIGSEGV);
1730017         proc_sig_term (pid, SIGPIPE);
1730018         proc_sig_term (pid, SIGALRM);
1730019         proc_sig_term (pid, SIGTERM);
1730020         proc_sig_term (pid, SIGUSR1);
1730021         proc_sig_term (pid, SIGUSR2);
1730022         proc_sig_chld (pid, SIGCHLD);
1730023         proc_sig_cont (pid, SIGCONT);
1730024         proc_sig_stop (pid, SIGSTOP);
1730025         proc_sig_stop (pid, SIGTSTP);
1730026         proc_sig_stop (pid, SIGTTIN);
1730027         proc_sig_stop (pid, SIGTTOU);
1730028       }
```

```
1730029 }
```

## kernel/proc/proc_sch_terminals.c

«

Si veda la sezione i159.8.9.

```
1740001 #include <kernel/proc.h>
1740002 #include <kernel/k_libc.h>
1740003 //---------------------------------------------------------------------
1740004 void
1740005 proc_sch_terminals (void)
1740006 {
1740007     pid_t  pid;
1740008     int    key;
1740009     tty_t *tty;
1740010     dev_t  device;
1740011     //
1740012     // Try to read a key from console keyboard buffer (only consoles
1740013     // are available).
1740014     //
1740015     key = con_char_ready ();
1740016     if (key == 0)
1740017       {
1740018         //
1740019         // No key is ready on the keyboard buffer: just return.
1740020         //
1740021         return;
1740022       }
1740023     //
1740024     // A key is available. Find the currently active console.
1740025     //
1740026     device = tty_console ((dev_t) 0);
1740027     tty    = tty_reference (device);
1740028     if (tty == NULL)
1740029       {
1740030         k_printf ("kernel alert: console device 0x%04x not found!\n",
1740031                   device);
1740032         //
1740033         // Will send the typed character to the first terminal!
1740034         //
1740035         tty = tty_reference ((dev_t) 0);
1740036       }
1740037     //
1740038     // Defined the active console. Put the character there.
1740039     //
1740040     if (tty->key == 0)
1740041       {
1740042         tty->status = TTY_OK;
1740043       }
1740044     else
1740045       {
1740046         tty->status = TTY_LOST_KEY;
1740047       }
1740048     tty->key = con_char_read ();
1740049     //
1740050     // Verify if it is a control key that must be handled. If so, a
1740051     // signal is sent to all processes with the same control terminal,
1740052     // excluded the kernel (0) and 'init' (1). Such control keys are not
1740053     // passed to the applications.
1740054     //
1740055     // Please note that this a simplified solution, because the signal
1740056     // should reach only the foreground process of the group. For that
1740057     // reason, only che [Ctrl C] is taken into consideration, because
1740058     // processes can ignore the signal 'SIGINT'.
1740059     //
1740060     if (tty->pgrp != 0)
1740061       {
1740062         //
1740063         // There is a process group for that terminal.
1740064         //
1740065         if      (tty->key == 3)         // [Ctrl C] -> SIGINT
1740066           {
1740067             for (pid = 2; pid <  PROCESS_MAX; pid++)
1740068               {
1740069                 if (proc_table[pid].pgrp == tty->pgrp)
1740070                   {
1740071                     k_kill (pid, SIGINT);
1740072                   }
1740073               }
1740074             tty->key    = 0;    // Reset key and status.
1740075             tty->status = TTY_OK;
1740076           }
1740077       }
1740078     //
1740079     // Check for a console switch key combination.
1740080     //
1740081     if      (tty->key == 0x11)          // [Ctrl Q] -> DC1 -> console0.
1740082       {
1740083         tty->key    = 0;                // Reset key and status.
1740084         tty->status = TTY_OK;
1740085         tty_console (DEV_CONSOLE0);     // Switch.
1740086       }
1740087     else if (tty->key == 0x12)          // [Ctrl R] -> DC2 -> console1.
1740088       {
1740089         tty->key    = 0;                // Reset key and status.
1740090         tty->status = TTY_OK;
1740091         tty_console (DEV_CONSOLE1);     // Switch.
1740092       }
1740093     else if (tty->key == 0x13)          // [Ctrl S] -> DC3 -> console2.
1740094       {
1740095         tty->key    = 0;                // Reset key and status.
```



```
1740096         tty->status = TTY_OK;
1740097         tty_console (DEV_CONSOLE2);     // Switch.
1740098       }
1740099     else if (tty->key == 0x14)          // [Ctrl T] -> DC4 -> console3.
1740100       {
1740101         tty->key    = 0;                // Reset key and status.
1740102         tty->status = TTY_OK;
1740103         tty_console (DEV_CONSOLE3);     // Switch.
1740104       }
1740105     //
1740106     // A key was pressed: must wake up all processes waiting for reading
1740107     // a terminal: all processes must be reactivated, because a process
1740108     // can read from the device file, and not just from its own
1740109     // terminal.
1740110     //
1740111     for (pid = 0; pid < PROCESS_MAX; pid++)
1740112       {
1740113         if (   (proc_table[pid].status == PROC_SLEEPING)
1740114             && (proc_table[pid].wakeup_events & WAKEUP_EVENT_TTY))
1740115           {
1740116             //
1740117             // A process waiting for that terminal was found:
1740118             // remove the waiting event and set it ready.
1740119             //
1740120             proc_table[pid].wakeup_events &= ~WAKEUP_EVENT_TTY;
1740121             proc_table[pid].status = PROC_READY;
1740122           }
1740123       }
1740124 }
```

## kernel/proc/proc_sch_timers.c

«

Si veda la sezione i159.8.10.

```
1750001 #include <kernel/proc.h>
1750002 #include <kernel/k_libc.h>
1750003 //---------------------------------------------------------------------
1750004 void
1750005 proc_sch_timers (void)
1750006 {
1750007     static unsigned long int previous_time;
1750008            unsigned long int current_time;
1750009            unsigned int      pid;
1750010     current_time = k_time (NULL);
1750011     if (previous_time != current_time)
1750012       {
1750013         for (pid = 0; pid < PROCESS_MAX; pid++)
1750014           {
1750015             if (   (proc_table[pid].wakeup_events & WAKEUP_EVENT_TIMER)
1750016                 && (proc_table[pid].status == PROC_SLEEPING)
1750017                 && (proc_table[pid].wakeup_timer > 0))
1750018               {
1750019                 proc_table[pid].wakeup_timer--;
1750020                 if (proc_table[pid].wakeup_timer == 0)
1750021                   {
1750022                     proc_table[pid].status = PROC_READY;
1750023                   }
1750024               }
1750025           }
1750026       }
1750027     previous_time = current_time;
1750028 }
```

## kernel/proc/proc_scheduler.c

«

Si veda la sezione i159.8.11.

```
1760001 #include <kernel/proc.h>
1760002 #include <kernel/k_libc.h>
1760003 #include <stdint.h>
1760004 //---------------------------------------------------------------------
1760005 extern uint16_t _ksp;
1760006 //---------------------------------------------------------------------
1760007 void
1760008 proc_scheduler (uint16_t *sp, segment_t *segment_d)
1760009 {
1760010     //
1760011     // The process is identified from the data and stack segment.
1760012     //
1760013     pid_t prev;
1760014     pid_t next;
1760015     //
1760016     static unsigned long int previous_clock;
1760017            unsigned long int current_clock;
1760018     //
1760019     // Check if current data segments are right.
1760020     //
1760021     if (es () != ds () || ss () != ds ())
1760022       {
1760023         k_printf ("\n");
1760024         k_printf ("Kernel panic: ES, DS, SS are different!\n");
1760025         k_exit (0);
1760026       }
1760027     //
1760028     // Search the data segment inside the process table.
1760029     // Must be done here, because the subsequent call to
1760030     // proc_sch_signals() will remove the segment numbers
1760031     // from a zombie process.
1760032     //
1760033     prev = proc_find (*segment_d);
```

```
1760034     //
1760035     // Take care of sleeping processes: wake up if sleeping time
1760036     // elapsed.
1760037     //
1760038     proc_sch_timers ();
1760039     //
1760040     // Take care of pending signals.
1760041     //
1760042     proc_sch_signals ();
1760043     //
1760044     // Take care input from terminals.
1760045     //
1760046     proc_sch_terminals ();
1760047     //
1760048     // Update the CPU time usage.
1760049     //
1760050     current_clock   = k_clock ();
1760051     proc_table[prev].usage += current_clock - previous_clock;
1760052     previous_clock  = current_clock;
1760053     //
1760054     // Scan for a next process.
1760055     //
1760056     for (next = prev+1; next != prev; next++)
1760057       {
1760058         if (next >= PROCESS_MAX)
1760059           {
1760060             next = -1; // At the next loop, 'next' will be zero.
1760061             continue;
1760062           }
1760063         if      (proc_table[next].status == PROC_EMPTY)
1760064           {
1760065             continue;
1760066           }
1760067         else if (proc_table[next].status == PROC_CREATED)
1760068           {
1760069             continue;
1760070           }
1760071         else if (proc_table[next].status == PROC_READY)
1760072           {
1760073             if (proc_table[prev].status == PROC_RUNNING)
1760074               {
1760075                 proc_table[prev].status = PROC_READY;
1760076               }
1760077             proc_table[prev].sp     = *sp;
1760078             proc_table[next].status = PROC_RUNNING;
1760079             proc_table[next].ret    = 0;
1760080             *segment_d      = proc_table[next].segment_d;
1760081             *sp             = proc_table[next].sp;
1760082             break;
1760083           }
1760084         else if (proc_table[next].status == PROC_RUNNING)
1760085           {
1760086             if (proc_table[prev].status == PROC_RUNNING)
1760087               {
1760088                 k_printf ("Kernel alert: process %i ", prev);
1760089                 k_printf ("and %i \"running\"!\r", next);
1760090                 proc_table[prev].status = PROC_READY;
1760091               }
1760092             proc_table[prev].sp     = *sp;
1760093             proc_table[next].status = PROC_RUNNING;
1760094             proc_table[next].ret    = 0;
1760095             *segment_d      = proc_table[next].segment_d;
1760096             *sp             = proc_table[next].sp;
1760097             break;
1760098           }
1760099         else if (proc_table[next].status == PROC_SLEEPING)
1760100           {
1760101             continue;
1760102           }
1760103         else if (proc_table[next].status == PROC_ZOMBIE)
1760104           {
1760105             continue;
1760106           }
1760107       }
1760108     //
1760109     // Check again if the next process is set to running, otherwise set
1760110     // the kernel to such value!
1760111     //
1760112     next = proc_find (*segment_d);
1760113     if (proc_table[next].status != PROC_RUNNING)
1760114       {
1760115         proc_table[0].status = PROC_RUNNING;
1760116         *segment_d   = proc_table[0].segment_d;
1760117         *sp          = proc_table[0].sp;
1760118       }
1760119     //
1760120     // Save kernel stack pointer.
1760121     //
1760122     _ksp = proc_table[0].sp;
1760123     //
1760124     // At the end, must inform the PIC 1, with message «EOI».
1760125     //
1760126     out_8 (0x20, 0x20);
1760127 }
```



## kernel/proc/proc_sig_chld.c

«

Si veda la sezione i159.8.12.

```
1770001 #include <kernel/proc.h>
1770002 //---------------------------------------------------------------------
1770003 void
1770004 proc_sig_chld (pid_t parent, int sig)
1770005 {
1770006     pid_t child;
1770007     //
1770008     // Please note that 'sig' should be SIGCHLD and nothing else.
1770009     // So, the following test, means to verify if the parent process
1770010     // has received a SIGCHLD already.
1770011     //
1770012     if (proc_sig_status (parent, sig))
1770013       {
1770014         if (   (!proc_sig_ignore (parent, sig))
1770015             && (proc_table[parent].status == PROC_SLEEPING)
1770016             && (proc_table[parent].wakeup_events & WAKEUP_EVENT_SIGNAL)
1770017             && (proc_table[parent].wakeup_signal == sig))
1770018           {
1770019             //
1770020             // The signal is not ignored from the parent process;
1770021             // the parent process is sleeping;
1770022             // the parent process is waiting for a signal;
1770023             // the parent process is waiting for current signal.
1770024             // So, just wake it up.
1770025             //
1770026             proc_table[parent].status         = PROC_READY;
1770027             proc_table[parent].wakeup_events ^= WAKEUP_EVENT_SIGNAL;
1770028             proc_table[parent].wakeup_signal  = 0;
1770029           }
1770030         else
1770031           {
1770032             //
1770033             // All other cases, means to remove all dead children.
1770034             //
1770035             for (child = 1; child < PROCESS_MAX; child++)
1770036               {
1770037                 if (   proc_table[child].ppid == parent
1770038                     && proc_table[child].status == PROC_ZOMBIE)
1770039                   {
1770040                     proc_available (child);
1770041                   }
1770042               }
1770043           }
1770044         proc_sig_off (parent, sig);
1770045       }
1770046 }
```

## kernel/proc/proc_sig_cont.c

«

Si veda la sezione i159.8.13.

```
1780001 #include <kernel/proc.h>
1780002 //---------------------------------------------------------------------
1780003 void
1780004 proc_sig_cont (pid_t pid, int sig)
1780005 {
1780006     //
1780007     // The value for argument 'sig' should be SIGCONT.
1780008     //
1780009     if (proc_sig_status (pid, sig))
1780010       {
1780011         if (proc_sig_ignore (pid, sig))
1780012           {
1780013             proc_sig_off (pid, sig);
1780014           }
1780015         else
1780016           {
1780017             proc_table[pid].status = PROC_READY;
1780018             proc_sig_off (pid, sig);
1780019           }
1780020       }
1780021 }
```

## kernel/proc/proc_sig_core.c

«

Si veda la sezione i159.8.14.

```
1790001 #include <kernel/proc.h>
1790002 //---------------------------------------------------------------------
1790003 void
1790004 proc_sig_core (pid_t pid, int sig)
1790005 {
1790006     addr_t address_i;
1790007     addr_t address_d;
1790008     size_t size_i;
1790009     size_t size_d;
1790010     //
1790011     if (proc_sig_status (pid, sig))
1790012       {
1790013         if (proc_sig_ignore (pid, sig))
1790014           {
1790015             proc_sig_off (pid, sig);
1790016           }
1790017         else
1790018           {
1790019             //
```

```
1790020             // Save process addresses and sizes (might be useful if
1790021             // we want to try to exit the process before core dump.
1790022             //
1790023             address_i   = proc_table[pid].address_i;
1790024             address_d   = proc_table[pid].address_d;
1790025             size_i      = proc_table[pid].size_i;
1790026             size_d      = proc_table[pid].size_d;
1790027             //
1790028             // Core dump: the process who formally writes the file
1790029             // is the terminating one.
1790030             //
1790031             if (address_d == address_i)
1790032               {
1790033                 proc_dump_memory (pid, address_i, size_i, "core");
1790034               }
1790035             else
1790036               {
1790037                 proc_dump_memory (pid, address_i, size_i, "core.i");
1790038                 proc_dump_memory (pid, address_d, size_d, "core.d");
1790039               }
1790040             //
1790041             // The signal, translated to negative, is returned (but
1790042             // the effective value received by the application will
1790043             // be cutted, leaving only the low 8 bit).
1790044             //
1790045             proc_sys_exit (pid, -sig);
1790046           }
1790047       }
1790048 }
```

## kernel/proc/proc_sig_ignore.c

«

Si veda la sezione i159.8.15.

```
1800001 #include <kernel/proc.h>
1800002 //-------------------------------------------------------------------------
1800003 int
1800004 proc_sig_ignore (pid_t pid, int sig)
1800005 {
1800006     unsigned long int flag = 1L << (sig - 1);
1800007     if (proc_table[pid].sig_ignore & flag)
1800008       {
1800009         return (1);
1800010       }
1800011     else
1800012       {
1800013         return (0);
1800014       }
1800015 }
```

## kernel/proc/proc_sig_off.c

«

Si veda la sezione i159.8.16.

```
1810001 #include <kernel/proc.h>
1810002 //-------------------------------------------------------------------------
1810003 void
1810004 proc_sig_off (pid_t pid, int sig)
1810005 {
1810006     unsigned long int flag = 1L << (sig - 1);
1810007     proc_table[pid].sig_status ^= flag;
1810008 }
```

## kernel/proc/proc_sig_on.c

«

Si veda la sezione i159.8.16.

```
1820001 #include <kernel/proc.h>
1820002 //-------------------------------------------------------------------------
1820003 void
1820004 proc_sig_on (pid_t pid, int sig)
1820005 {
1820006     unsigned long int flag = 1L << (sig - 1);
1820007     proc_table[pid].sig_status |= flag;
1820008 }
```

## kernel/proc/proc_sig_status.c

«

Si veda la sezione i159.8.17.

```
1830001 #include <kernel/proc.h>
1830002 //-------------------------------------------------------------------------
1830003 int
1830004 proc_sig_status (pid_t pid, int sig)
1830005 {
1830006     unsigned long int flag = 1L << (sig - 1);
1830007     if (proc_table[pid].sig_status & flag)
1830008       {
1830009         return (1);
1830010       }
1830011     else
1830012       {
1830013         return (0);
1830014       }
1830015 }
```



## kernel/proc/proc_sig_stop.c

«

Si veda la sezione i159.8.18.

```
1840001 #include <kernel/proc.h>
1840002 //-------------------------------------------------------------------------
1840003 void
1840004 proc_sig_stop (pid_t pid, int sig)
1840005 {
1840006     if (proc_sig_status (pid, sig))
1840007       {
1840008         if (proc_sig_ignore (pid, sig) && !(sig == SIGSTOP))
1840009           {
1840010             proc_sig_off (pid, sig);
1840011           }
1840012         else
1840013           {
1840014             proc_table[pid].status = PROC_SLEEPING;
1840015             proc_table[pid].ret    = -sig;
1840016             proc_sig_off (pid, sig);
1840017           }
1840018       }
1840019 }
```

## kernel/proc/proc_sig_term.c

«

Si veda la sezione i159.8.19.

```
1850001 #include <kernel/proc.h>
1850002 //-------------------------------------------------------------------------
1850003 void
1850004 proc_sig_term (pid_t pid, int sig)
1850005 {
1850006     if (proc_sig_status (pid, sig))
1850007       {
1850008         if (proc_sig_ignore (pid, sig) && !(sig == SIGKILL))
1850009           {
1850010             proc_sig_off (pid, sig);
1850011           }
1850012         else
1850013           {
1850014             //
1850015             // The signal, translated to negative, is returned (but
1850016             // the effective value received by the application will
1850017             // be cutted, leaving only the low 8 bit).
1850018             //
1850019             proc_sys_exit (pid, -sig);
1850020           }
1850021       }
1850022 }
```

## kernel/proc/proc_sys_exec.c

«

Si veda la sezione i159.8.20.

```
1860001 #include <kernel/proc.h>
1860002 #include <errno.h>
1860003 #include <fcntl.h>
1860004 //-------------------------------------------------------------------------
1860005 int
1860006 proc_sys_exec (uint16_t *sp, segment_t *segment_d, pid_t pid,
1860007                const char *path,
1860008                unsigned int argc, char *arg_data,
1860009                unsigned int envc, char *env_data)
1860010 {
1860011   unsigned int      i;
1860012   unsigned int      j;
1860013   char             *arg;
1860014   char             *env;
1860015   char             *envp[ARG_MAX/16];
1860016   char             *argv[ARG_MAX/16];
1860017   size_t            size;
1860018   size_t            arg_data_size;
1860019   size_t            env_data_size;
1860020   unsigned int      p_off;
1860021   inode_t          *inode;
1860022   ssize_t           size_read;
1860023   header_t          header;
1860024   unsigned long int s;        // Help calculating process sizes.
1860025   uint16_t          new_sp;
1860026   uint16_t          envp_address;
1860027   uint16_t          argv_address;
1860028   size_t            process_size_i;
1860029   size_t            process_size_d;
1860030   char              buffer[MEM_BLOCK_SIZE];
1860031   uint16_t          stack_element;
1860032   off_t             inode_start;
1860033   addr_t            memory_start;
1860034   addr_t            previous_address_i;
1860035   segment_t         previous_segment_i;
1860036   size_t            previous_size_i;
1860037   addr_t            previous_address_d;
1860038   segment_t         previous_segment_d;
1860039   size_t            previous_size_d;
1860040   int               status;
1860041   memory_t          allocated_i;
1860042   memory_t          allocated_d;
1860043   pid_t             extra;
1860044   int               proc_count;
1860045   file_t           *file;
```

```
  int               fdn;
  dev_t             device;
  int               eof;
  //
  // Check for limits.
  //
  if (argc > (ARG_MAX/16) || envc > (ARG_MAX/16))
    {
      errset (ENOMEM);
      return (-1);
    }
  //
  // Scan arguments to calculate the full size and the relative
  // pointers. The final size is rounded to 2, for the stack.
  //
  arg = arg_data;
  for (i = 0, j = 0; i < argc; i++)
    {
      argv[i] = (char *) j;   // Relative pointer inside
                              // the 'arg_data'.
      size = strlen (arg);
      arg += size + 1;
      j += size + 1;
    }
  arg_data_size = j;
  if (arg_data_size % 2)
    {
      arg_data_size++;
    }
  //
  // Scan environment variables to calculate the full size and the
  // relative pointers. The final size is rounded to 2, for the stack.
  //
  env = env_data;
  for (i = 0, j = 0; i < envc; i++)
    {
      envp[i] = (char *) j;   // Relative pointer inside
                              // the 'env_data'.
      size = strlen (env);
      env += size + 1;
      j += size + 1;
    }
  env_data_size = j;
  if (env_data_size % 2)
    {
      env_data_size++;
    }
  //
  // Read the inode related to the executable file name.
  // Function path_inode() includes the inode get procedure.
  //
  inode = path_inode (pid, path);
  if (inode == NULL)
    {
      errset (ENOENT);        // No such file or directory.
      return (-1);
    }
  //
  // Check for permissions.
  //
  status = inode_check (inode, S_IFREG, 5, proc_table[pid].euid);
  if (status != 0)
    {
      //
      // File is not of a valid type or permission are not
      // sufficient: release the executable file inode
      // and return with an error.
      //
      inode_put (inode);
      errset (EACCES);        // Permission denied.
      return (-1);
    }
  //
  // Read the header from the executable file.
  //
  size_read = inode_file_read (inode, (off_t) 0, &header,
                               (sizeof header), &eof);
  if (size_read != (sizeof header))
    {
      //
      // The file is shorter than the executable header, so, it isn't
      // an executable: release the file inode and return with an
      // error.
      //
      inode_put (inode);
      errset (ENOEXEC);
      return (-1);
    }
  if (header.magic0 != MAGIC_OS16 || header.magic1 != MAGIC_OS16_APPL)
    {
      //
      // The header does not have the expected magic numbers, so,
      // it isn't a valid executable: release the file inode and
      // return with an error.
      //
      inode_put (inode);
      errset (ENOEXEC);
      return (-1);    // This is not a valid executable!
    }
  //
  // Calculate data size.
```



```
  //
  s  = header.ebss;
  if (header.ssize == 0)
    {
      s += 0x10000L;  // Zero means max size.
    }
  else
    {
      s += header.ssize;
    }
  if (s > 0xFFFF)
    {
      process_size_d = 0x0000;  // 0x0000 means the maximum size:
                                // 0x10000.
      new_sp         = 0x0000;  // 0x0000 is like 0x10000 and the first
                                // push moves SP to 0xFFFE.
    }
  else
    {
      process_size_d = s;
      new_sp         = process_size_d;
      if (new_sp % 2)
        {
          new_sp--;       // The stack pointer should be even.
        }
    }
  //
  // Calculate code size.
  //
  if (header.segoff == 0)
    {
      process_size_i = process_size_d;
    }
  else
    {
      process_size_i = header.segoff * 16;
    }
  //
  // Allocate memory: code and data segments.
  //
  status = mb_alloc_size (process_size_i, &allocated_i);
  if (status < 0)
    {
      //
      // The program instructions (code segment) cannot be loaded
      // into memory: release the executable file inode and return
      // with an error.
      //
      inode_put (inode);
      errset (ENOMEM);        // Not enough space.
      return ((pid_t) -1);
    }
  if (header.segoff == 0)
    {
      //
      // Code and data segments are the same: no need
      // to allocate more memory for the data segment.
      //
      allocated_d.address = allocated_i.address;
      allocated_d.segment = allocated_i.segment;
      allocated_d.size    = allocated_i.size;
    }
  else
    {
      //
      // Code and data segments are different: the data
      // segment memory is allocated.
      //
      status = mb_alloc_size (process_size_d, &allocated_d);
      if (status < 0)
        {
          //
          // The separated program data (data segment) cannot be loaded
          // into memory: free the already allocated memory for the
          // program instructions, release the executable file inode
          // and return with an error.
          //
          mb_free (allocated_i.address, allocated_i.size);
          inode_put (inode);
          errset (ENOMEM);        // Not enough space.
          return ((pid_t) -1);
        }
    }
  //
  // Load executable in memory.
  //
  if (header.segoff == 0)
    {
      //
      // Code and data share the same segment.
      //
      for (eof = 0, memory_start = allocated_i.address,
           inode_start = 0, size_read = 0;
           inode_start < inode->size && !eof;
           inode_start += size_read)
        {
          memory_start += size_read;
          //
          // Read a block of memory.
          //
          size_read = inode_file_read (inode, inode_start,
```

```
                                    buffer, MEM_BLOCK_SIZE, &eof);
          if (size_read < 0)
            {
              //
              // Free memory and inode.
              //
              mb_free (allocated_i.address, allocated_i.size);
              inode_put (inode);
              errset (EIO);
              return (-1);
            }
          //
          // Copy inside the right position to be executed.
          //
          dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE, memory_start, buffer,
                  (size_t) size_read, NULL);
        }
    }
  else
    {
      //
      // Code and data with different segments.
      //
      for (eof = 0, memory_start = allocated_i.address,
           inode_start = 0, size_read = 0;
           inode_start < process_size_i && !eof;
           inode_start += size_read)
        {
          memory_start += size_read;
          //
          // Read a block of memory
          //
          size_read = inode_file_read (inode, inode_start,
                                       buffer, MEM_BLOCK_SIZE, &eof);
          if (size_read < 0)
            {
              //
              // Free memory and inode.
              //
              mb_free (allocated_i.address, allocated_i.size);
              mb_free (allocated_d.address, allocated_d.size);
              inode_put (inode);
              errset (EIO);
              return (-1);
            }
          //
          // Copy inside the right position to be executed.
          //
          dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE, memory_start, buffer,
                  (size_t) size_read, NULL);
        }
      for (eof = 0, memory_start = allocated_d.address,
           inode_start = (header.segoff * 16), size_read = 0;
           inode_start < inode->size && !eof;
           inode_start += size_read)
        {
          memory_start += size_read;
          //
          // Read a block of memory
          //
          size_read = inode_file_read (inode, inode_start,
                                       buffer, MEM_BLOCK_SIZE, &eof);
          if (size_read < 0)
            {
              //
              // Free memory and inode.
              //
              mb_free (allocated_i.address, allocated_i.size);
              mb_free (allocated_d.address, allocated_d.size);
              inode_put (inode);
              errset (EIO);
              return (-1);
            }
          dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE, memory_start, buffer,
                  (size_t) size_read, NULL);
        }
    }
  //
  // The executable file was successfully loaded in memory:
  // release the executable file inode.
  //
  inode_put (inode);
  //
  // Put environment data inside the stack.
  //
  new_sp -= env_data_size; //------------------------------ environment
  mem_copy (address (seg_d (), (unsigned int) env_data),
            (allocated_d.address + new_sp), env_data_size);
  //
  // Put arguments data inside the stack.
  //
  new_sp -= arg_data_size; //-------------------------------- arguments
  mem_copy (address (seg_d (), (unsigned int) arg_data),
            (allocated_d.address + new_sp), arg_data_size);
  //
  // Put envp[] inside the stack, updating all the pointers.
  //
  new_sp -= 2; //------------------------------------------------ NULL
  stack_element = NULL;
  dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
          (allocated_d.address + new_sp),
```



```
          &stack_element, (sizeof stack_element), NULL);
  //
  p_off  = new_sp;            //
  p_off += 2;                 // Calculate memory pointers from
  p_off += arg_data_size;     // original relative pointers,
  for (i = 0; i < envc; i++)  // inside the environment array
    {                         // of pointers.
      envp[i] += p_off;       //
    }                         //
  //
  new_sp -= (envc * (sizeof (char *))); //---------------------- *envp[]
  mem_copy (address (seg_d (), (unsigned int) envp),
            (allocated_d.address + new_sp),
            (envc * (sizeof (char *))));
  //
  // Save the envp[] location, needed in the following.
  //
  envp_address = new_sp;
  //
  // Put argv[] inside the stack, updating all the pointers.
  //
  new_sp -= 2; //------------------------------------------------ NULL
  stack_element = NULL;
  dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
          (allocated_d.address + new_sp),
          &stack_element, (sizeof stack_element), NULL);
  //
  p_off  = new_sp;                      //
  p_off += 2 ;                          // Calculate memory pointers
  p_off += (envc * (sizeof (char *)));  // from original relative
  p_off += 2;                           // pointers, inside the
  for (i = 0; i < argc; i++)            // arguments array of
    {                                   // pointers.
      argv[i] += p_off;                 //
    }                                   //
  //
  new_sp -= (argc * (sizeof (char *))); //---------------------- *argv[]
  mem_copy (address (seg_d (), (unsigned int) argv),
            (allocated_d.address + new_sp),
            (argc * (sizeof (char *))));
  //
  // Save the argv[] location, needed in the following.
  //
  argv_address = new_sp;
  //
  // Put the pointer to the array envp[].
  //
  new_sp -= 2; //------------------------------------------------ argc
  stack_element = envp_address;
  dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
          (allocated_d.address + new_sp),
          &stack_element, (sizeof stack_element), NULL);
  //
  // Put the pointer to the array argv[].
  //
  new_sp -= 2; //------------------------------------------------ argc
  stack_element = argv_address;
  dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
          (allocated_d.address + new_sp),
          &stack_element, (sizeof stack_element), NULL);
  //
  // Put argc inside the stack.
  //
  new_sp -= 2; //------------------------------------------------ argc
  dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
          (allocated_d.address + new_sp),
          &argc, (sizeof argc), NULL);
  //
  // Set the rest of the stack.
  //
  new_sp -= 2; //----------------------------------------------- FLAGS
  stack_element = 0x0200;
  dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
          (allocated_d.address + new_sp),
          &stack_element, (sizeof stack_element), NULL);
  new_sp -= 2; //------------------------------------------------- CS
  dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
          (allocated_d.address + new_sp),
          &allocated_i.segment, (sizeof allocated_i.segment), NULL);
  new_sp -= 2; //------------------------------------------------ IP
  stack_element = 0;
  dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
          (allocated_d.address + new_sp),
          &stack_element, (sizeof stack_element), NULL);
  new_sp -= 2; //------------------------------------------------ ES
  dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
          (allocated_d.address + new_sp),
          &allocated_d.segment, (sizeof allocated_d.segment), NULL);
  new_sp -= 2; //------------------------------------------------- DS
  dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
          (allocated_d.address + new_sp),
          &allocated_d.segment, (sizeof allocated_d.segment), NULL);
  new_sp -= 2; //------------------------------------------------ DI
  stack_element = 0;
  dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
          (allocated_d.address + new_sp),
          &stack_element, (sizeof stack_element), NULL);
  new_sp -= 2; //------------------------------------------------ SI
  stack_element = 0;
  dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
          (allocated_d.address + new_sp),
```

```
1860450           &stack_element, (sizeof stack_element), NULL);
1860451   new_sp -= 2; //----------------------------------------------------- BP
1860452   stack_element = 0;
1860453   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860454           (allocated_d.address + new_sp),
1860455           &stack_element, (sizeof stack_element), NULL);
1860456   new_sp -= 2; //----------------------------------------------------- BX
1860457   stack_element = 0;
1860458   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860459           (allocated_d.address + new_sp),
1860460           &stack_element, (sizeof stack_element), NULL);
1860461   new_sp -= 2; //----------------------------------------------------- DX
1860462   stack_element = 0;
1860463   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860464           (allocated_d.address + new_sp),
1860465           &stack_element, (sizeof stack_element), NULL);
1860466   new_sp -= 2; //----------------------------------------------------- CX
1860467   stack_element = 0;
1860468   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860469           (allocated_d.address + new_sp),
1860470           &stack_element, (sizeof stack_element), NULL);
1860471   new_sp -= 2; //----------------------------------------------------- AX
1860472   stack_element = 0;
1860473   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1860474           (allocated_d.address + new_sp),
1860475           &stack_element, (sizeof stack_element), NULL);
1860476   //
1860477   // Close process file descriptors, if the 'FD_CLOEXEC' flag
1860478   // is present.
1860479   //
1860480   for (fdn = 0; fdn < OPEN_MAX; fdn++)
1860481     {
1860482       if (proc_table[pid].fd[0].file != NULL)
1860483         {
1860484           if (proc_table[pid].fd[0].fd_flags & FD_CLOEXEC)
1860485             {
1860486               fd_close (pid, fdn);
1860487             }
1860488         }
1860489     }
1860490   //
1860491   // Select device for standard I/O, if a standard I/O stream must be
1860492   // opened.
1860493   //
1860494   if (proc_table[pid].device_tty != 0)
1860495     {
1860496       device = proc_table[pid].device_tty;
1860497     }
1860498   else
1860499     {
1860500       device = DEV_TTY;
1860501     }
1860502   //
1860503   // Prepare missing standard file descriptors. The function
1860504   // 'file_stdio_dev_make()' arranges the value for 'errno' if
1860505   // necessary. If a standard file descriptor cannot be allocated,
1860506   // the program is left without it.
1860507   //
1860508   if (proc_table[pid].fd[0].file == NULL)
1860509     {
1860510       file = file_stdio_dev_make (device, S_IFCHR, O_RDONLY);
1860511       if (file != NULL)                                    // stdin
1860512         {
1860513           proc_table[pid].fd[0].fl_flags     = O_RDONLY;
1860514           proc_table[pid].fd[0].fd_flags     = 0;
1860515           proc_table[pid].fd[0].file         = file;
1860516           proc_table[pid].fd[0].file->offset = 0;
1860517         }
1860518     }
1860519   if (proc_table[pid].fd[1].file == NULL)
1860520     {
1860521       file = file_stdio_dev_make (device, S_IFCHR, O_WRONLY);
1860522       if (file != NULL)                                   // stdout
1860523         {
1860524           proc_table[pid].fd[1].fl_flags     = O_WRONLY;
1860525           proc_table[pid].fd[1].fd_flags     = 0;
1860526           proc_table[pid].fd[1].file         = file;
1860527           proc_table[pid].fd[1].file->offset = 0;
1860528         }
1860529     }
1860530   if (proc_table[pid].fd[2].file == NULL)
1860531     {
1860532       file = file_stdio_dev_make (device, S_IFCHR, O_WRONLY);
1860533       if (file != NULL)                                   // stderr
1860534         {
1860535           proc_table[pid].fd[2].fl_flags     = O_WRONLY;
1860536           proc_table[pid].fd[2].fd_flags     = 0;
1860537           proc_table[pid].fd[2].file         = file;
1860538           proc_table[pid].fd[2].file->offset = 0;
1860539         }
1860540     }
1860541   //
1860542   // Prepare to switch
1860543   //
1860544   previous_address_i = proc_table[pid].address_i;
1860545   previous_segment_i = proc_table[pid].segment_i;
1860546   previous_size_i    = proc_table[pid].size_i;
1860547   previous_address_d = proc_table[pid].address_d;
1860548   previous_segment_d = proc_table[pid].segment_d;
1860549   previous_size_d    = proc_table[pid].size_d;
1860550   //
```



```
1860551   proc_table[pid].address_i  = allocated_i.address;
1860552   proc_table[pid].segment_i  = allocated_i.segment;
1860553   proc_table[pid].size_i     = allocated_i.size;
1860554   proc_table[pid].address_d  = allocated_d.address;
1860555   proc_table[pid].segment_d  = allocated_d.segment;
1860556   proc_table[pid].size_d     = allocated_d.size;
1860557   proc_table[pid].sp         = new_sp;
1860558   strncpy (proc_table[pid].name, path, PATH_MAX);
1860559   //
1860560   // Ensure to have a terminated string.
1860561   //
1860562   proc_table[pid].name[PATH_MAX-1] = 0;
1860563   //
1860564   // Free data segment memory.
1860565   //
1860566   mb_free (previous_address_d, previous_size_d);
1860567   //
1860568   // Free code segment memory if it is
1860569   // different from the data segment.
1860570   //
1860571   if (previous_segment_i != previous_segment_d)
1860572     {
1860573       //
1860574       // Must verify if no other process is
1860575       // using the same memory.
1860576       //
1860577       for (proc_count = 0, extra = 0; extra < PROCESS_MAX; extra++)
1860578         {
1860579           if (proc_table[extra].status == PROC_EMPTY   ||
1860580               proc_table[extra].status == PROC_ZOMBIE)
1860581             {
1860582               continue;
1860583             }
1860584           if (previous_segment_i == proc_table[extra].segment_i)
1860585             {
1860586               proc_count++;
1860587             }
1860588         }
1860589       if (proc_count == 0)
1860590         {
1860591           //
1860592           // The code segment can be released, because no other
1860593           // process is using it.
1860594           //
1860595           mb_free (previous_address_i, previous_size_i);
1860596         }
1860597     }
1860598   //
1860599   // Change the segment and the stack pointer, from the interrupt.
1860600   //
1860601   *segment_d = proc_table[pid].segment_d;
1860602   *sp        = proc_table[pid].sp;
1860603   //
1860604   return (0);
1860605 }
```

## kernel/proc/proc_sys_exit.c

«

Si veda la sezione i159.8.21.

```
1870001 #include <kernel/proc.h>
1870002 #include <kernel/k_libc.h>
1870003 //---------------------------------------------------------------------
1870004 void
1870005 proc_sys_exit (pid_t pid, int status)
1870006 {
1870007   pid_t  child;
1870008   pid_t  parent = proc_table[pid].ppid;
1870009   pid_t  extra;
1870010   int    proc_count;
1870011   int    sigchld = 0;
1870012   int    fdn;
1870013   tty_t *tty;
1870014   //
1870015   proc_table[pid].status     = PROC_ZOMBIE;
1870016   proc_table[pid].ret        = status;
1870017   proc_table[pid].sig_status =  0;
1870018   proc_table[pid].sig_ignore =  0;
1870019   //
1870020   // Close files.
1870021   //
1870022   for (fdn = 0; fdn < OPEN_MAX; fdn++)
1870023     {
1870024       fd_close (pid, fdn);
1870025     }
1870026   //
1870027   // Close current directory.
1870028   //
1870029   inode_put (proc_table[pid].inode_cwd);
1870030   //
1870031   // Close the controlling terminal, if it is a process leader with
1870032   // such a terminal.
1870033   //
1870034   if (proc_table[pid].pgrp == pid && proc_table[pid].device_tty != 0)
1870035     {
1870036       tty = tty_reference (proc_table[pid].device_tty);
1870037       //
1870038       // Verify.
1870039       //
1870040       if (tty == NULL)
1870041         {
```

```
1870042           //
1870043           // Show a kernel message.
1870044           //
1870045           k_printf ("kernel alert: cannot find the terminal item "
1870046                     "for device 0x%04x!\n",
1870047                     (int) proc_table[pid].device_tty);
1870048         }
1870049       else if (tty->pgrp != pid)
1870050         {
1870051           //
1870052           // Show a kernel message.
1870053           //
1870054           k_printf ("kernel alert: terminal device 0x%04x should "
1870055                     "be associated to the process group %i, but it "
1870056                     "is instead related to process group %i!\n",
1870057                     (int) proc_table[pid].device_tty, (int) pid,
1870058                     (int) tty->pgrp);
1870059         }
1870060       else
1870061         {
1870062           tty->pgrp = 0;
1870063         }
1870064     }
1870065   //
1870066   // Free data segment memory.
1870067   //
1870068   mb_free (proc_table[pid].address_d, proc_table[pid].size_d);
1870069   //
1870070   // Free code segment memory if it is
1870071   // different from the data segment.
1870072   //
1870073   if (proc_table[pid].segment_i != proc_table[pid].segment_d)
1870074     {
1870075       //
1870076       // Must verify if no other process is using the same memory.
1870077       // The proc_count variable is incremented for processes
1870078       // active, ready or sleeping: the current process is already
1870079       // set as zombie, and is not counted.
1870080       //
1870081       for (proc_count = 0, extra = 0; extra < PROCESS_MAX; extra++)
1870082         {
1870083           if (proc_table[extra].status == PROC_EMPTY   ||
1870084               proc_table[extra].status == PROC_ZOMBIE)
1870085             {
1870086               continue;
1870087             }
1870088           if (proc_table[pid].segment_i == proc_table[extra].segment_i)
1870089             {
1870090               proc_count++;
1870091             }
1870092         }
1870093       if (proc_count == 0)
1870094         {
1870095           //
1870096           // The code segment can be released, because no other
1870097           // process, except the current one (to be closed),
1870098           // is using it.
1870099           //
1870100           mb_free (proc_table[pid].address_i, proc_table[pid].size_i);
1870101         }
1870102     }
1870103   //
1870104   // Abandon children to 'init' ((pid_t) 1).
1870105   //
1870106   for (child = 1; child < PROCESS_MAX; child++)
1870107     {
1870108       if (   proc_table[child].status != PROC_EMPTY
1870109           && proc_table[child].ppid == pid)
1870110         {
1870111           proc_table[child].ppid = 1; // Son of 'init'.
1870112           if (proc_table[child].status == PROC_ZOMBIE)
1870113             {
1870114               sigchld = 1;    // Must send a SIGCHLD to 'init'.
1870115             }
1870116         }
1870117     }
1870118   //
1870119   // SIGCHLD to 'init'.
1870120   //
1870121   if (   sigchld
1870122       && pid != 1
1870123       && proc_table[1].status != PROC_EMPTY
1870124       && proc_table[1].status != PROC_ZOMBIE)
1870125     {
1870126       proc_sig_on ((pid_t) 1, SIGCHLD);
1870127     }
1870128   //
1870129   // Announce to the parent the death of its child.
1870130   //
1870131   if (   pid != parent
1870132       && proc_table[parent].status != PROC_EMPTY)
1870133     {
1870134       proc_sig_on (parent, SIGCHLD);
1870135     }
1870136 }
```





Si veda la sezione i159.8.22.

```
1880001 #include <kernel/proc.h>
1880002 #include <errno.h>
1880003 //---------------------------------------------------------------------
1880004 pid_t
1880005 proc_sys_fork (pid_t ppid, uint16_t sp)
1880006 {
1880007     pid_t     pid;
1880008     pid_t     zombie;
1880009     memory_t  allocated_i;
1880010     memory_t  allocated_d;
1880011     int       status;
1880012     int       fdn;
1880013     //
1880014     // Find a free PID.
1880015     //
1880016     for (pid = 1; pid < PROCESS_MAX; pid++)
1880017       {
1880018         if (proc_table[pid].status == PROC_EMPTY)
1880019           {
1880020             break;
1880021           }
1880022       }
1880023     if (pid >= PROCESS_MAX)
1880024       {
1880025         //
1880026         // There is no free pid.
1880027         //
1880028         errset (ENOMEM);        // Not enough space.
1880029         return (-1);
1880030       }
1880031     //
1880032     // Before allocating a new process, must check if there are some
1880033     // zombie slots, still with original segment data: should reset
1880034     // it now!
1880035     //
1880036     for (zombie = 1; zombie < PROCESS_MAX; zombie++)
1880037       {
1880038         if (   proc_table[zombie].status == PROC_ZOMBIE
1880039             && proc_table[zombie].segment_d != -1)
1880040           {
1880041             proc_table[zombie].segment_i = -1;    // Reset
1880042             proc_table[zombie].address_i = -1L;   // memory
1880043             proc_table[zombie].size_i    =  0;    // allocation
1880044             proc_table[zombie].segment_d = -1;    // data
1880045             proc_table[zombie].address_d = -1L;   // to
1880046             proc_table[zombie].size_d    =  0;    // impossible
1880047             proc_table[zombie].sp        =  0;    // values.
1880048           }
1880049       }
1880050     //
1880051     // Allocate memory: code and data segments.
1880052     //
1880053     if (proc_table[ppid].segment_i == proc_table[ppid].segment_d)
1880054       {
1880055         //
1880056         // Code segment and Data segment are the same
1880057         // (same I&D).
1880058         //
1880059         status = mb_alloc_size (proc_table[ppid].size_i, &allocated_i);
1880060         if (status < 0)
1880061           {
1880062             errset (ENOMEM);        // Not enough space.
1880063             return ((pid_t) -1);
1880064           }
1880065         allocated_d.address = allocated_i.address;
1880066         allocated_d.segment = allocated_i.segment;
1880067         allocated_d.size    = allocated_i.size;
1880068       }
1880069     else
1880070       {
1880071         //
1880072         // Code segment and Data segment are different
1880073         // (different I&D).
1880074         // Only the data segment is allocated.
1880075         //
1880076         status = mb_alloc_size (proc_table[ppid].size_d, &allocated_d);
1880077         if (status < 0)
1880078           {
1880079             errset (ENOMEM);        // Not enough space.
1880080             return ((pid_t) -1);
1880081           }
1880082         //
1880083         // Code segment is the same from the parent process.
1880084         //
1880085         allocated_i.address = proc_table[ppid].address_i;
1880086         allocated_i.segment = proc_table[ppid].segment_i;
1880087         allocated_i.size    = proc_table[ppid].size_i;
1880088       }
1880089     //
1880090     // Copy the process in memory.
1880091     //
1880092     if (proc_table[ppid].segment_i == proc_table[ppid].segment_d)
1880093       {
1880094         //
1880095         // Code segment and data segment are the same:
1880096         // must copy all.
1880097         //
1880098         // Copy the code segment: if the size is zero,
```

```
1880099         // it means 0x10000 bytes (65536 bytes).
1880100         //
1880101         if (proc_table[ppid].size_i == 0)
1880102           {
1880103             //
1880104             // Copy 0x10000 bytes with two steps.
1880105             //
1880106             mem_copy (proc_table[ppid].address_i,
1880107                       allocated_i.address, 0x8000);
1880108             mem_copy ((proc_table[ppid].address_i + 0x8000),
1880109                       (allocated_i.address + 0x8000), 0x8000);
1880110           }
1880111         else
1880112           {
1880113             //
1880114             // Normal copy.
1880115             //
1880116             mem_copy (proc_table[ppid].address_i, allocated_i.address,
1880117                       proc_table[ppid].size_i);
1880118           }
1880119       }
1880120     else
1880121       {
1880122         //
1880123         // Code segment and data segment are different:
1880124         // copy only the data segment.
1880125         //
1880126         // Copy the data segment in memory: if the size is zero,
1880127         // it means 0x10000 bytes (65536 bytes).
1880128         //
1880129         if (proc_table[ppid].size_d == 0)
1880130           {
1880131             //
1880132             // Copy 0x10000 bytes with two steps.
1880133             //
1880134             mem_copy (proc_table[ppid].address_d,
1880135                       allocated_d.address, 0x8000);
1880136             mem_copy ((proc_table[ppid].address_d + 0x8000),
1880137                       (allocated_d.address + 0x8000), 0x8000);
1880138           }
1880139         else
1880140           {
1880141             //
1880142             // Normal copy.
1880143             //
1880144             mem_copy (proc_table[ppid].address_d, allocated_d.address,
1880145                       proc_table[ppid].size_d);
1880146           }
1880147       }
1880148     //
1880149     // Allocate the new PID.
1880150     //
1880151     proc_table[pid].ppid          = ppid;
1880152     proc_table[pid].pgrp          = proc_table[ppid].pgrp;
1880153     proc_table[pid].uid           = proc_table[ppid].uid;
1880154     proc_table[pid].euid          = proc_table[ppid].euid;
1880155     proc_table[pid].suid          = proc_table[ppid].suid;
1880156     proc_table[pid].device_tty    = proc_table[ppid].device_tty;
1880157     proc_table[pid].sig_status    = 0;
1880158     proc_table[pid].sig_ignore    = 0;
1880159     proc_table[pid].usage         = 0;
1880160     proc_table[pid].status        = PROC_CREATED;
1880161     proc_table[pid].wakeup_events = 0;
1880162     proc_table[pid].wakeup_signal = 0;
1880163     proc_table[pid].wakeup_timer  = 0;
1880164     proc_table[pid].segment_i     = allocated_i.segment;
1880165     proc_table[pid].address_i     = allocated_i.address;
1880166     proc_table[pid].size_i        = proc_table[ppid].size_i;
1880167     proc_table[pid].segment_d     = allocated_d.segment;
1880168     proc_table[pid].address_d     = allocated_d.address;
1880169     proc_table[pid].size_d        = proc_table[ppid].size_d;
1880170     proc_table[pid].sp            = sp;
1880171     proc_table[pid].ret           = 0;
1880172     proc_table[pid].inode_cwd     = proc_table[ppid].inode_cwd;
1880173     proc_table[pid].umask         = proc_table[ppid].umask;
1880174     strncpy (proc_table[pid].name,
1880175              proc_table[ppid].name, PATH_MAX);
1880176     strncpy (proc_table[pid].path_cwd,
1880177              proc_table[ppid].path_cwd, PATH_MAX);
1880178     //
1880179     // Increase inode references for the working directory.
1880180     //
1880181     proc_table[pid].inode_cwd->references++;
1880182     //
1880183     // Duplicate valid file descriptors.
1880184     //
1880185     for (fdn = 0; fdn < OPEN_MAX; fdn++)
1880186       {
1880187         if (   proc_table[ppid].fd[fdn].file != NULL
1880188             && proc_table[ppid].fd[fdn].file->inode != NULL)
1880189           {
1880190             //
1880191             // Copy to the forked process.
1880192             //
1880193             proc_table[pid].fd[fdn].fl_flags
1880194                 = proc_table[ppid].fd[fdn].fl_flags;
1880195             proc_table[pid].fd[fdn].fd_flags
1880196                 = proc_table[ppid].fd[fdn].fd_flags;
1880197             proc_table[pid].fd[fdn].file
1880198                 = proc_table[ppid].fd[fdn].file;
1880199             //
```



```
1880200             // Increment file reference.
1880201             //
1880202             proc_table[ppid].fd[fdn].file->references++;
1880203           }
1880204       }
1880205     //
1880206     // Change segment values inside the stack: DS==ES; CS.
1880207     //
1880208     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1880209             (allocated_d.address + proc_table[pid].sp + 14),
1880210             &allocated_d.segment, (sizeof allocated_d.segment), NULL);
1880211     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1880212             (allocated_d.address + proc_table[pid].sp + 16),
1880213             &allocated_d.segment, (sizeof allocated_d.segment), NULL);
1880214     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1880215             (allocated_d.address + proc_table[pid].sp + 20),
1880216             &allocated_i.segment, (sizeof allocated_i.segment), NULL);
1880217     //
1880218     // Set it ready.
1880219     //
1880220     proc_table[pid].status  = PROC_READY;
1880221     //
1880222     // Return the new PID.
1880223     //
1880224     return (pid);
1880225 }
```

## kernel/proc/proc_sys_kill.c

Si veda la sezione i159.8.23.

```
1890001 #include <kernel/proc.h>
1890002 #include <errno.h>
1890003 //---------------------------------------------------------------------
1890004 int
1890005 proc_sys_kill (pid_t pid_killer, pid_t pid_target, int sig)
1890006 {
1890007     uid_t euid = proc_table[pid_killer].euid;
1890008     uid_t uid  = proc_table[pid_killer].uid;
1890009     pid_t pgrp = proc_table[pid_killer].pgrp;
1890010     int   p;                    // Index inside the process table.
1890011     //
1890012     if (pid_target < -1)
1890013       {
1890014         errset (ESRCH);
1890015         return (-1);
1890016       }
1890017     else if (pid_target == -1)
1890018       {
1890019         if (sig == 0)
1890020           {
1890021             return (0);
1890022           }
1890023         if (euid == 0)
1890024           {
1890025             //
1890026             // Because 'pid_target' is qual to '-1' and the effective
1890027             // user identity is '0', then, all processes,
1890028             // except the kernel and init, will receive the signal.
1890029             //
1890030             // The following scan starts from 2, to preserve the
1890031             // kernel and init processes.
1890032             //
1890033             for (p = 2; p < PROCESS_MAX; p++)
1890034               {
1890035                 if (   proc_table[p].status != PROC_EMPTY
1890036                     && proc_table[p].status != PROC_ZOMBIE)
1890037                   {
1890038                     proc_sig_on (p, sig);
1890039                   }
1890040               }
1890041           }
1890042         else
1890043           {
1890044             //
1890045             // Because 'pid_target' is qual to '-1', but the effective
1890046             // user identity is not '0', then, all processes owned
1890047             // by the same effective user identity, will receive the
1890048             // signal.
1890049             //
1890050             // The following scan starts from 1, to preserve the
1890051             // kernel process.
1890052             //
1890053             for (p = 1; p < PROCESS_MAX; p++)
1890054               {
1890055                 if (   proc_table[p].status != PROC_EMPTY
1890056                     && proc_table[p].status != PROC_ZOMBIE
1890057                     && proc_table[p].uid == euid)
1890058                   {
1890059                     proc_sig_on (p, sig);
1890060                   }
1890061               }
1890062           }
1890063         return (0);
1890064       }
1890065     else if (pid_target == 0)
1890066       {
1890067         if (sig == 0)
1890068           {
1890069             return (0);
1890070           }
```

```
1890071         //
1890072         // The following scan starts from 1, to preserve the
1890073         // kernel process.
1890074         //
1890075         for (p = 1; p < PROCESS_MAX; p++)
1890076           {
1890077             if (   proc_table[p].status != PROC_EMPTY
1890078                 && proc_table[p].status != PROC_ZOMBIE
1890079                 && proc_table[p].pgrp == pgrp)
1890080               {
1890081                 proc_sig_on (p, sig);
1890082               }
1890083           }
1890084         return (0);
1890085       }
1890086     else if (pid_target >= PROCESS_MAX)
1890087       {
1890088         errset (ESRCH);
1890089         return (-1);
1890090       }
1890091     else // (pid_target > 0)
1890092       {
1890093         if (proc_table[pid_target].status == PROC_EMPTY  ||
1890094             proc_table[pid_target].status == PROC_ZOMBIE)
1890095           {
1890096             errset (ESRCH);
1890097             return (-1);
1890098           }
1890099         else if (uid  == proc_table[pid_target].uid   ||
1890100                  uid  == proc_table[pid_target].suid  ||
1890101                  euid == proc_table[pid_target].uid   ||
1890102                  euid == proc_table[pid_target].suid  ||
1890103                  euid == 0)
1890104           {
1890105             if (sig == 0)
1890106               {
1890107                 return (0);
1890108               }
1890109             else
1890110               {
1890111                 proc_sig_on (pid_target, sig);
1890112                 return (0);
1890113               }
1890114           }
1890115         else
1890116           {
1890117             errset (EPERM);
1890118             return (-1);
1890119           }
1890120       }
1890121 }
```

kernel/proc/proc_sys_seteuid.c

«

Si veda la sezione i159.8.24.

```
1900001 #include <kernel/proc.h>
1900002 #include <errno.h>
1900003 //-----------------------------------------------------------------------
1900004 int
1900005 proc_sys_seteuid (pid_t pid, uid_t euid)
1900006 {
1900007   if (proc_table[pid].euid == 0)
1900008     {
1900009       proc_table[pid].euid = euid;
1900010       return (0);
1900011     }
1900012   else if (euid == proc_table[pid].euid)
1900013     {
1900014       return (0);
1900015     }
1900016   else if (euid == proc_table[pid].uid || euid == proc_table[pid].suid)
1900017     {
1900018       proc_table[pid].euid = euid;
1900019       return (0);
1900020     }
1900021   else
1900022     {
1900023       errset (EPERM);
1900024       return (-1);
1900025     }
1900026 }
```

kernel/proc/proc_sys_setuid.c

«

Si veda la sezione i159.8.25.

```
1910001 #include <kernel/proc.h>
1910002 #include <errno.h>
1910003 //-----------------------------------------------------------------------
1910004 int
1910005 proc_sys_setuid (pid_t pid, uid_t uid)
1910006 {
1910007     if (proc_table[pid].euid == 0)
1910008       {
1910009         proc_table[pid].uid  = uid;
1910010         proc_table[pid].euid = uid;
1910011         proc_table[pid].suid = uid;
1910012         return (0);
1910013       }
```



```
1910014     else if (uid == proc_table[pid].euid)
1910015       {
1910016         return (0);
1910017       }
1910018     else if (uid == proc_table[pid].uid || uid == proc_table[pid].suid)
1910019       {
1910020         proc_table[pid].euid = uid;
1910021         return (0);
1910022       }
1910023     else
1910024       {
1910025         errset (EPERM);
1910026         return (-1);
1910027       }
1910028 }
```

kernel/proc/proc_sys_signal.c

«

Si veda la sezione i159.8.26.

```
1920001 #include <kernel/proc.h>
1920002 #include <errno.h>
1920003 //-----------------------------------------------------------------------
1920004 sighandler_t
1920005 proc_sys_signal (pid_t pid, int sig, sighandler_t handler)
1920006 {
1920007     unsigned long int flag = 1L << (sig - 1);
1920008     sighandler_t      previous;
1920009
1920010     if (sig <= 0)
1920011       {
1920012         errset (EINVAL);
1920013         return (SIG_ERR);
1920014       }
1920015
1920016     if (proc_table[pid].sig_ignore & flag)
1920017       {
1920018         previous = SIG_IGN;
1920019       }
1920020     else
1920021       {
1920022         previous = SIG_DFL;
1920023       }
1920024
1920025     if (handler == SIG_DFL)
1920026       {
1920027         proc_table[pid].sig_ignore ^= flag;
1920028         return (previous);
1920029       }
1920030     else if (handler == SIG_IGN)
1920031       {
1920032         proc_table[pid].sig_ignore |= flag;
1920033         return (previous);
1920034       }
1920035     else
1920036       {
1920037         errset (EINVAL);
1920038         return (SIG_ERR);
1920039       }
1920040 }
```

kernel/proc/proc_sys_wait.c

«

Si veda la sezione i159.8.27.

```
1930001 #include <kernel/proc.h>
1930002 #include <errno.h>
1930003 //-----------------------------------------------------------------------
1930004 pid_t
1930005 proc_sys_wait (pid_t pid, int *status)
1930006 {
1930007     pid_t  parent = pid;
1930008     pid_t  child;
1930009     int    child_available = 0;
1930010     //
1930011     // Find a dead child process.
1930012     //
1930013     for (child = 1; child < PROCESS_MAX; child++)
1930014       {
1930015         if (proc_table[child].ppid == parent)
1930016           {
1930017             child_available = 1;        // Child found!
1930018             if (proc_table[child].status == PROC_ZOMBIE)
1930019               {
1930020                 break;                  // It is dead!
1930021               }
1930022           }
1930023       }
1930024     //
1930025     // If the index 'child' is a valid process number,
1930026     // a dead child was found.
1930027     //
1930028     if (child < PROCESS_MAX)
1930029       {
1930030         *status = proc_table[child].ret;
1930031         proc_available (child);
1930032         return (child);
1930033       }
1930034     else
1930035       {
```

```
1930036         if (child_available)
1930037           {
1930038             //
1930039             // There are child, but all alive.
1930040             //
1930041             // Go to sleep.
1930042             //
1930043             proc_table[parent].status         = PROC_SLEEPING;
1930044             proc_table[parent].wakeup_events |= WAKEUP_EVENT_SIGNAL;
1930045             proc_table[parent].wakeup_signal  = SIGCHLD;
1930046             return ((pid_t) 0);
1930047           }
1930048         else
1930049           {
1930050             //
1930051             // There are no child at all.
1930052             //
1930053             errset (ECHILD);
1930054             return ((pid_t) -1);
1930055           }
1930056       }
1930057 }
```

## kernel/proc/proc_table.c

«

Si veda la sezione i159.8.7.

```
1940001 #include <kernel/proc.h>
1940002 //---------------------------------------------------------------------------
1940003 proc_t   proc_table[PROCESS_MAX];
```

## kernel/proc/sysroutine.c

«

Si veda la sezione i159.8.28.

```
1950001 #include <kernel/proc.h>
1950002 #include <errno.h>
1950003 #include <kernel/k_libc.h>
1950004 //---------------------------------------------------------------------------
1950005 static void sysroutine_error_back (int *number, int *line,
1950006                                    char *file_name);
1950007 //---------------------------------------------------------------------------
1950008 void
1950009 sysroutine (uint16_t *sp, segment_t *segment_d, uint16_t syscallnr,
1950010             uint16_t msg_off, uint16_t msg_size)
1950011 {
1950012   pid_t  pid      = proc_find (*segment_d);
1950013   addr_t msg_addr = address (*segment_d, msg_off);
1950014   //
1950015   // Inbox.
1950016   //
1950017   union {
1950018       sysmsg_chdir_t     chdir;
1950019       sysmsg_chmod_t     chmod;
1950020       sysmsg_chown_t     chown;
1950021       sysmsg_clock_t     clock;
1950022       sysmsg_close_t     close;
1950023       sysmsg_dup_t       dup;
1950024       sysmsg_dup2_t      dup2;
1950025       sysmsg_exec_t      exec;
1950026       sysmsg_exit_t      exit;
1950027       sysmsg_fchmod_t    fchmod;
1950028       sysmsg_fchown_t    fchown;
1950029       sysmsg_fcntl_t     fcntl;
1950030       sysmsg_fork_t      fork;
1950031       sysmsg_fstat_t     fstat;
1950032       sysmsg_kill_t      kill;
1950033       sysmsg_link_t      link;
1950034       sysmsg_lseek_t     lseek;
1950035       sysmsg_mkdir_t     mkdir;
1950036       sysmsg_mknod_t     mknod;
1950037       sysmsg_mount_t     mount;
1950038       sysmsg_open_t      open;
1950039       sysmsg_read_t      read;
1950040       sysmsg_seteuid_t   seteuid;
1950041       sysmsg_setuid_t    setuid;
1950042       sysmsg_signal_t    signal;
1950043       sysmsg_sleep_t     sleep;
1950044       sysmsg_stat_t      stat;
1950045       sysmsg_stime_t     stime;
1950046       sysmsg_time_t      time;
1950047       sysmsg_uarea_t     uarea;
1950048       sysmsg_umask_t     umask;
1950049       sysmsg_umount_t    umount;
1950050       sysmsg_unlink_t    unlink;
1950051       sysmsg_wait_t      wait;
1950052       sysmsg_write_t     write;
1950053       sysmsg_zpchar_t    zpchar;
1950054       sysmsg_zpstring_t  zpstring;
1950055   } msg;
1950056   //
1950057   // Verify if the system call was emitted from kernel code.
1950058   // The kernel can emit only some particular system call:
1950059   //   SYS_NULL, to let other processes run;
1950060   //   SYS_FORK, to let fork itself;
1950061   //   SYS_EXEC, to replace a forked copy of itself.
1950062   //
1950063   if (pid == 0)
1950064     {
1950065       //
```



```
1950066       // This is the kernel code!
1950067       //
1950068       if (   syscallnr != SYS_0
1950069           && syscallnr != SYS_FORK
1950070           && syscallnr != SYS_EXEC)
1950071         {
1950072           k_printf ("kernel panic: the system call %i ", syscallnr);
1950073           k_printf ("was received while running in kernel space!\n");
1950074         }
1950075     }
1950076   //
1950077   // Entering a system call, the kernel variable 'errno' must be
1950078   // reset, otherwise, a previous kernel code error might be returned
1950079   // to the applications.
1950080   //
1950081   errno    = 0;
1950082   errln    = 0;
1950083   errfn[0] = 0;
1950084   //
1950085   // Get message.
1950086   //
1950087   dev_io (pid, DEV_MEM, DEV_READ, msg_addr, &msg, msg_size, NULL);
1950088   //
1950089   // Do the request from the received system call.
1950090   //
1950091   switch (syscallnr)
1950092     {
1950093       case SYS_0:
1950094           break;
1950095       case SYS_CHDIR:
1950096           msg.chdir.ret         = path_chdir (pid, msg.chdir.path);
1950097           sysroutine_error_back (&msg.chdir.errno, &msg.chdir.errln,
1950098                                  msg.chdir.errfn);
1950099           break;
1950100       case SYS_CHMOD:
1950101           msg.chmod.ret         = path_chmod (pid, msg.chmod.path,
1950102                                               msg.chmod.mode);
1950103           sysroutine_error_back (&msg.chmod.errno, &msg.chmod.errln,
1950104                                  msg.chmod.errfn);
1950105           break;
1950106       case SYS_CHOWN:
1950107           msg.chown.ret         = path_chown (pid, msg.chown.path,
1950108                                               msg.chown.uid,
1950109                                               msg.chown.gid);
1950110           sysroutine_error_back (&msg.chown.errno, &msg.chown.errln,
1950111                                  msg.chown.errfn);
1950112           break;
1950113       case SYS_CLOCK:
1950114           msg.clock.ret         = k_clock ();
1950115           break;
1950116       case SYS_CLOSE:
1950117           msg.close.ret         = fd_close (pid, msg.close.fdn);
1950118           sysroutine_error_back (&msg.close.errno, &msg.close.errln,
1950119                                  msg.close.errfn);
1950120           break;
1950121       case SYS_DUP:
1950122           msg.dup.ret           = fd_dup (pid, msg.dup.fdn_old, 0);
1950123           sysroutine_error_back (&msg.dup.errno, &msg.dup.errln,
1950124                                  msg.dup.errfn);
1950125           break;
1950126       case SYS_DUP2:
1950127           msg.dup2.ret          = fd_dup2 (pid, msg.dup2.fdn_old,
1950128                                            msg.dup2.fdn_new);
1950129           sysroutine_error_back (&msg.dup2.errno, &msg.dup2.errln,
1950130                                  msg.dup2.errfn);
1950131           break;
1950132       case SYS_EXEC:
1950133           msg.exec.ret          = proc_sys_exec (sp, segment_d, pid,
1950134                                                  msg.exec.path,
1950135                                                  msg.exec.argc,
1950136                                                  msg.exec.arg_data,
1950137                                                  msg.exec.envc,
1950138                                                  msg.exec.env_data);
1950139           msg.exec.uid          = proc_table[pid].uid;
1950140           msg.exec.euid         = proc_table[pid].euid;
1950141           sysroutine_error_back (&msg.exec.errno, &msg.exec.errln,
1950142                                  msg.exec.errfn);
1950143           break;
1950144       case SYS_EXIT:
1950145           if (pid == 0)
1950146             {
1950147               k_printf ("kernel alert: "
1950148                         "the kernel cannot exit!\n");
1950149             }
1950150           else
1950151             {
1950152               proc_sys_exit (pid, msg.exit.status);
1950153             }
1950154           break;
1950155       case SYS_FCHMOD:
1950156           msg.fchmod.ret        = fd_chmod (pid, msg.fchmod.fdn,
1950157                                             msg.fchmod.mode);
1950158           sysroutine_error_back (&msg.fchmod.errno, &msg.fchmod.errln,
1950159                                  msg.fchmod.errfn);
1950160           break;
1950161       case SYS_FCHOWN:
1950162           msg.fchown.ret        = fd_chown (pid, msg.fchown.fdn,
1950163                                             msg.fchown.uid,
1950164                                             msg.fchown.gid);
1950165           sysroutine_error_back (&msg.fchown.errno, &msg.fchown.errln,
1950166                                  msg.fchown.errfn);
```

```
1950167             break;
1950168         case SYS_FCNTL:
1950169             msg.fcntl.ret         = fd_fcntl (pid, msg.fcntl.fdn,
1950170                                               msg.fcntl.cmd,
1950171                                               msg.fcntl.arg);
1950172             sysroutine_error_back (&msg.fcntl.errno, &msg.fcntl.errln,
1950173                                    msg.fcntl.errfn);
1950174             break;
1950175         case SYS_FORK:
1950176             msg.fork.ret          = proc_sys_fork (pid, *sp);
1950177             sysroutine_error_back (&msg.fork.errno, &msg.fork.errln,
1950178                                    msg.fork.errfn);
1950179             break;
1950180         case SYS_FSTAT:
1950181             msg.fstat.ret         = fd_stat (pid, msg.fstat.fdn,
1950182                                              &msg.fstat.stat);
1950183             sysroutine_error_back (&msg.fstat.errno, &msg.fstat.errln,
1950184                                    msg.fstat.errfn);
1950185             break;
1950186         case SYS_KILL:
1950187             msg.kill.ret          = proc_sys_kill (pid, msg.kill.pid,
1950188                                                    msg.kill.signal);
1950189             sysroutine_error_back (&msg.kill.errno, &msg.kill.errln,
1950190                                    msg.kill.errfn);
1950191             break;
1950192         case SYS_LINK:
1950193             msg.link.ret          = path_link (pid, msg.link.path_old,
1950194                                                msg.link.path_new);
1950195             sysroutine_error_back (&msg.link.errno, &msg.link.errln,
1950196                                    msg.link.errfn);
1950197             break;
1950198         case SYS_LSEEK:
1950199             msg.lseek.ret         = fd_lseek (pid, msg.lseek.fdn,
1950200                                               msg.lseek.offset,
1950201                                               msg.lseek.whence);
1950202             sysroutine_error_back (&msg.lseek.errno, &msg.lseek.errln,
1950203                                    msg.lseek.errfn);
1950204             break;
1950205         case SYS_MKDIR:
1950206             msg.mkdir.ret         = path_mkdir (pid, msg.mkdir.path,
1950207                                                 msg.mkdir.mode);
1950208             sysroutine_error_back (&msg.mkdir.errno, &msg.mkdir.errln,
1950209                                    msg.mkdir.errfn);
1950210             break;
1950211         case SYS_MKNOD:
1950212             msg.mknod.ret         = path_mknod (pid, msg.mknod.path,
1950213                                                 msg.mknod.mode,
1950214                                                 msg.mknod.device);
1950215             sysroutine_error_back (&msg.mknod.errno, &msg.mknod.errln,
1950216                                    msg.mknod.errfn);
1950217             break;
1950218         case SYS_MOUNT:
1950219             msg.mount.ret         = path_mount (pid, msg.mount.path_dev,
1950220                                                 msg.mount.path_mnt,
1950221                                                 msg.mount.options);
1950222             sysroutine_error_back (&msg.mount.errno, &msg.mount.errln,
1950223                                    msg.mount.errfn);
1950224             break;
1950225         case SYS_OPEN:
1950226             msg.open.ret          = fd_open (pid, msg.open.path,
1950227                                              msg.open.flags,
1950228                                              msg.open.mode);
1950229             sysroutine_error_back (&msg.open.errno, &msg.open.errln,
1950230                                    msg.open.errfn);
1950231             break;
1950232         case SYS_PGRP:
1950233             proc_table[pid].pgrp          = pid;
1950234             break;
1950235         case SYS_READ:
1950236             msg.read.ret          = fd_read (pid, msg.read.fdn,
1950237                                              msg.read.buffer,
1950238                                              msg.read.count,
1950239                                              &msg.read.eof);
1950240             sysroutine_error_back (&msg.read.errno, &msg.read.errln,
1950241                                    msg.read.errfn);
1950242             break;
1950243         case SYS_SETEUID:
1950244             msg.seteuid.ret       = proc_sys_seteuid (pid,
1950245                                                       msg.seteuid.euid);
1950246             msg.seteuid.euid      = proc_table[pid].euid;
1950247             sysroutine_error_back (&msg.seteuid.errno, &msg.seteuid.errln,
1950248                                    msg.seteuid.errfn);
1950249             break;
1950250         case SYS_SETUID:
1950251             msg.setuid.ret        = proc_sys_setuid (pid,
1950252                                                      msg.setuid.euid);
1950253             msg.setuid.uid        = proc_table[pid].uid;
1950254             msg.setuid.euid       = proc_table[pid].euid;
1950255             msg.setuid.suid       = proc_table[pid].suid;
1950256             sysroutine_error_back (&msg.setuid.errno, &msg.setuid.errln,
1950257                                    msg.setuid.errfn);
1950258             break;
1950259         case SYS_SIGNAL:
1950260             msg.signal.ret        = proc_sys_signal (pid,
1950261                                                      msg.signal.signal,
1950262                                                     msg.signal.handler);
1950263             sysroutine_error_back (&msg.signal.errno, &msg.signal.errln,
1950264                                    msg.signal.errfn);
1950265             break;
1950266         case SYS_SLEEP:
1950267             proc_table[pid].status        = PROC_SLEEPING;
```



```
1950268             proc_table[pid].ret           = 0;
1950269             proc_table[pid].wakeup_events = msg.sleep.events;
1950270             proc_table[pid].wakeup_signal = msg.sleep.signal;
1950271             proc_table[pid].wakeup_timer  = msg.sleep.seconds;
1950272             break;
1950273         case SYS_STAT:
1950274             msg.stat.ret          = path_stat (pid, msg.stat.path,
1950275                                                &msg.stat.stat);
1950276             sysroutine_error_back (&msg.stat.errno, &msg.stat.errln,
1950277                                    msg.stat.errfn);
1950278             break;
1950279         case SYS_STIME:
1950280             msg.stime.ret         = k_stime (&msg.stime.timer);
1950281             break;
1950282         case SYS_TIME:
1950283             msg.time.ret          = k_time (NULL);
1950284             break;
1950285         case SYS_UAREA:
1950286             msg.uarea.uid         = proc_table[pid].uid;
1950287             msg.uarea.euid        = proc_table[pid].euid;
1950288             msg.uarea.pid         = pid;
1950289             msg.uarea.ppid        = proc_table[pid].ppid;
1950290             msg.uarea.pgrp        = proc_table[pid].pgrp;
1950291             msg.uarea.umask       = proc_table[pid].umask;
1950292             strncpy (msg.uarea.path_cwd,
1950293                      proc_table[pid].path_cwd, PATH_MAX);
1950294             break;
1950295         case SYS_UMASK:
1950296             msg.umask.ret         = proc_table[pid].umask;
1950297             proc_table[pid].umask = (msg.umask.umask & 00777);
1950298             break;
1950299         case SYS_UMOUNT:
1950300             msg.umount.ret        = path_umount (pid,
1950301                                                  msg.umount.path_mnt);
1950302             sysroutine_error_back (&msg.umount.errno, &msg.umount.errln,
1950303                                    msg.umount.errfn);
1950304             break;
1950305         case SYS_UNLINK:
1950306             msg.unlink.ret        = path_unlink (pid, msg.unlink.path);
1950307             sysroutine_error_back (&msg.unlink.errno, &msg.unlink.errln,
1950308                                    msg.unlink.errfn);
1950309             break;
1950310         case SYS_WAIT:
1950311             msg.wait.ret          = proc_sys_wait (pid,
1950312                                                    &msg.wait.status);
1950313             sysroutine_error_back (&msg.wait.errno, &msg.wait.errln,
1950314                                    msg.wait.errfn);
1950315             break;
1950316         case SYS_WRITE:
1950317             msg.write.ret         = fd_write (pid, msg.write.fdn,
1950318                                               msg.write.buffer,
1950319                                               msg.write.count);
1950320             sysroutine_error_back (&msg.write.errno, &msg.write.errln,
1950321                                    msg.write.errfn);
1950322             break;
1950323         case SYS_ZPCHAR:
1950324             dev_io (pid, DEV_TTY, DEV_WRITE, 0L, &msg.zpchar.c,
1950325                     1, NULL);
1950326             break;
1950327         case SYS_ZPSTRING:
1950328             dev_io (pid, DEV_TTY, DEV_WRITE, 0L,
1950329                     msg.zpstring.string,
1950330                     strlen (msg.zpstring.string), NULL);
1950331             break;
1950332         default:
1950333             k_printf ("kernel alert: unknown system call %i!\n",
1950334                       syscallnr);
1950335             break;
1950336       }
1950337     //
1950338     // Return value with a message back.
1950339     //
1950340     dev_io (pid, DEV_MEM, DEV_WRITE, msg_addr, &msg, msg_size, NULL);
1950341     //
1950342     // Continue with the scheduler.
1950343     //
1950344     proc_scheduler (sp, segment_d);
1950345   }
1950346   //---------------------------------------------------------------------
1950347   static void
1950348   sysroutine_error_back (int *number, int *line, char *file_name)
1950349   {
1950350     *number = errno;
1950351     *line   = errln;
1950352     strncpy (file_name, errfn, PATH_MAX);
1950353     file_name[PATH_MAX-1] = 0;
1950354   }
```

## os16: «kernel/tty.h»

Si veda la sezione u0.9.

```
1960001   #ifndef _KERNEL_TTY_H
1960002   #define _KERNEL_TTY_H  1
1960003
1960004   #include <stddef.h>
1960005   #include <stdint.h>
1960006   #include <sys/types.h>
1960007   #include <kernel/ibm_i86.h>
1960008
1960009   //---------------------------------------------------------------------
```

```
1960010 #define         TTYS_CONSOLE    IBM_I86_VIDEO_PAGES
1960011 #define         TTYS_SERIAL     0
1960012 #define         TTYS_TOTAL      (TTYS_CONSOLE + TTYS_SERIAL)
1960013 //---------------------------------------------------------------------
1960014 #define         TTY_OK          0
1960015 #define         TTY_LOST_KEY    1
1960016 //---------------------------------------------------------------------
1960017 typedef struct {
1960018     dev_t       device;
1960019     pid_t       pgrp;            // Process group.
1960020     int         key;             // Last pressed key.
1960021     int         status;          // 0 = ok, 1 = lost typed character.
1960022 } tty_t;
1960023 //---------------------------------------------------------------------
1960024 extern tty_t tty_table[TTYS_TOTAL];
1960025 //---------------------------------------------------------------------
1960026 tty_t *tty_reference (dev_t device);
1960027 dev_t  tty_console   (dev_t device);
1960028 int    tty_read      (dev_t device);
1960029 void   tty_write     (dev_t device, int c);
1960030 void   tty_init      (void);
1960031
1960032 #endif
```

## kernel/tty/tty_console.c

«

Si veda la sezione u0.9.

```
1970001 #include <sys/os16.h>
1970002 #include <kernel/tty.h>
1970003 //---------------------------------------------------------------------
1970004 dev_t
1970005 tty_console (dev_t device)
1970006 {
1970007     static dev_t device_active = DEV_CONSOLE0;  // First time.
1970008            dev_t device_previous;
1970009     //
1970010     // Check if it required only the current device.
1970011     //
1970012     if (device == 0)
1970013       {
1970014         return (device_active);
1970015       }
1970016     //
1970017     // Fix if the device is not valid.
1970018     //
1970019     if (device > DEV_CONSOLE3 || device < DEV_CONSOLE0)
1970020       {
1970021         device = DEV_CONSOLE0;
1970022       }
1970023     //
1970024     // Update.
1970025     //
1970026     device_previous = device_active;
1970027     device_active   = device;
1970028     //
1970029     // Switch.
1970030     //
1970031     con_select (device_active & 0x00FF);
1970032     //
1970033     // Return previous device value.
1970034     //
1970035     return (device_previous);
1970036 }
```

## kernel/tty/tty_init.c

«

Si veda la sezione u0.9.

```
1980001 #include <sys/os16.h>
1980002 #include <kernel/tty.h>
1980003 //---------------------------------------------------------------------
1980004 void
1980005 tty_init (void)
1980006 {
1980007     int page;   // console page.
1980008     //
1980009     // Console initialization: console pages correspond to the first
1980010     // terminal items.
1980011     //
1980012     for (page = 0; page < TTYS_CONSOLE; page++)
1980013       {
1980014         tty_table[page].device = DEV_CONSOLE0 + page;
1980015         tty_table[page].pgrp   = 0;
1980016         tty_table[page].key    = 0;
1980017         tty_table[page].status = TTY_OK;
1980018       }
1980019     //
1980020     // Set video mode.
1980021     //
1980022     con_init ();
1980023     //
1980024     // Select the first console.
1980025     //
1980026     tty_console (DEV_CONSOLE0);
1980027     //
1980028     // Nothing else to configure (only consoles are available).
1980029     //
1980030     return;
1980031 }
```



## kernel/tty/tty_read.c

«

Si veda la sezione u0.9.

```
1990001 #include <sys/os16.h>
1990002 #include <kernel/tty.h>
1990003 #include <kernel/k_libc.h>
1990004 //---------------------------------------------------------------------
1990005 int
1990006 tty_read (dev_t device)
1990007 {
1990008     tty_t   *tty;
1990009     int      key;
1990010     //
1990011     tty = tty_reference (device);
1990012     if (tty == NULL)
1990013       {
1990014         k_printf ("kernel alert: cannot find terminal device "
1990015                   "0x%04x!\n", (int) device);
1990016         //
1990017         return (0);
1990018       }
1990019     //
1990020     // Read key and remove from the source.
1990021     //
1990022     key      = tty->key;
1990023     tty->key = 0;
1990024     //
1990025     // Return the key.
1990026     //
1990027     return (key);
1990028
1990029 }
```

## kernel/tty/tty_reference.c

«

Si veda la sezione u0.9.

```
2000001 #include <kernel/tty.h>
2000002 //---------------------------------------------------------------------
2000003 tty_t *
2000004 tty_reference (dev_t device)
2000005 {
2000006     int t;              // Terminal index.
2000007     //
2000008     // If device is zero, a reference to the whole table is returned.
2000009     //
2000010     if (device == 0)
2000011       {
2000012         return (tty_table);
2000013       }
2000014     //
2000015     // Otherwise, a scan is made to find the selected device.
2000016     //
2000017     for (t = 0; t < TTYS_TOTAL; t++)
2000018       {
2000019         if (tty_table[t].device == device)
2000020           {
2000021             //
2000022             // Device found. Return the pointer.
2000023             //
2000024             return (&tty_table[t]);
2000025           }
2000026       }
2000027     //
2000028     // No device found!
2000029     //
2000030     return (NULL);
2000031 }
```

## kernel/tty/tty_table.c

«

Si veda la sezione u0.9.

```
2010001 #include <kernel/tty.h>
2010002 //---------------------------------------------------------------------
2010003 tty_t tty_table[TTYS_TOTAL];
```

## kernel/tty/tty_write.c

«

Si veda la sezione u0.9.

```
2020001 #include <sys/os16.h>
2020002 #include <kernel/tty.h>
2020003 //---------------------------------------------------------------------
2020004 void
2020005 tty_write (dev_t device, int c)
2020006 {
2020007     int   console;
2020008     //
2020009     if ((device & 0xFF00) == (DEV_CONSOLE_MAJOR << 8))
2020010       {
2020011         console = (device & 0x00FF);
2020012         con_putc (console, c);
2020013       }
2020014 }
```