94.1 os32: directory principale

94.1.1 applic.sep.ld

Si veda la sezione 84.1.3.

```
10001 /*************************************************
10002  * SEPARATED text from data
10003  *************************************************/
10004
10005 ENTRY (startup)
10006 SECTIONS {
10007     . = 0x0;
10008     _text_start = .;
10009     .text : {
10010         *(.text)
10011         . = ALIGN (0x4);
10012     }
10013     _text_end = .;
10014     . = 0x0;
10015     _data_start = .;
10016     .rodata : {
10017         *(.rodata)
10018         . = ALIGN (0x4);
10019     }
10020     .data : {
10021         *(.data)
10022         . = ALIGN (0x4);
10023     }
10024     _data_end = .;
10025     _bss_start = .;
10026     .bss : {
10027         *(COMMON)
10028         *(.bss)
10029         . = ALIGN (0x4);
10030     }
10031     _bss_end = .;
10032 }
```


94.1.2 bochs

Si veda la sezione 85.4.

```
20001 #!/bin/sh
20002
20003 if [ "$UID" = 0 ]
20004 then
20005   #   172.21.11.18                    172.21.11.16
20006   #  >----------point to point -----> >-------os32
20007   #   tap0 (linux)                    net1
20008   #
20009   # Dal lato Linux:
20010   #   ifconfig tap0 172.21.11.18 pointopoint \
20011   #             172.21.11.16 netmask 255.255.255.255
20012   #   route add -host 172.21.11.16 gw 172.21.11.18
20013   #
20014   # Dalla macchina 172.21.254.254:
20015   #   route add -host 172.21.11.16 gw 172.21.11.18
20016   #
20017   bochs -q \
20018     "boot: disk" \
20019     "ata0-master: type=disk, path=disk.hda" \
20020     "keyboard_mapping: enabled=1, \
20021        map=/usr/share/bochs/keymaps/x11-pc-it.map" \
20022     "keyboard_type: mf" \
20023     "vga: none" \
20024     "ne2k: mac=b0:c4:20:00:00:00, ioaddr=0x300, \
20025           irq=9, ethmod=tuntap, ethdev=/dev/net/tun, \
20026           script=./tap0" \
20027     "i440fxsupport: enabled=1, slot1=pcivga, \
20028                   slot2=ne2k" \
20029     "romimage: \
20030       file=\"/usr/share/bochs/BIOS-bochs-legacy\"" \
20031     "megs:128"
20032 else
20033   bochs -q \
20034     "boot: disk" \
20035     "ata0-master: type=disk, path=disk.hda" \
20036     "keyboard_mapping: enabled=1, \
20037        map=/usr/share/bochs/keymaps/x11-pc-it.map" \
20038     "keyboard_type: mf" \
20039     "vga: none" \
20040     "ne2k: mac=b0:c4:20:00:00:00, ioaddr=0x300, \
20041           irq=9, ethmod=null" \
20042     "i440fxsupport: enabled=1, slot1=pcivga, \
20043                    slot2=ne2k" \
20044     "romimage: \
20045       file=\"/usr/share/bochs/BIOS-bochs-legacy\"" \
20046     "megs:128"
20047 fi
```

94.1.3 elf-to-os32

Si veda la sezione 84.1.3.

```
30001 #!/bin/sh
30002 #
30003 #
30004 #
30005 g_sz ()
30006 {
30007       sed "s/^[[:space:]]*[0-9]*[[:space:]]*//" \
30008     | sed "s/\.[^[:space:]]*[[:space:]]*//" \
30009     | sed "s/\([0-9a-z]*\)[[:space:]].*$/\1/"
30010 }
30011 #
30012 g_vma ()
30013 {
30014       sed "s/^[[:space:]]*[0-9]*[[:space:]]*//" \
30015     | sed "s/\.[^[:space:]]*[[:space:]]*//" \
30016     | sed "s/[0-9a-f]*[[:space:]]*//" \
30017     | sed "s/\([0-9a-z]*\)[[:space:]].*$/\1/"
30018 }
30019 #
30020 g_st ()
30021 {
30022       sed "s/^[[:space:]]*[0-9]*[[:space:]]*//" \
30023     | sed "s/\.[^[:space:]]*[[:space:]]*//" \
30024     | sed "s/[0-9a-f]*[[:space:]]*//" \
30025     | sed "s/[0-9a-f]*[[:space:]]*//" \
30026     | sed "s/[0-9a-f]*[[:space:]]*//" \
30027     | sed "s/\([0-9a-z]*\)[[:space:]].*$/\1/"
30028 }
30029 #
30030 FILE_ELF="$1"
30031 FILE_OS32="$2"
30032 #
```

```
30033 if [ -e "$FILE_ELF" ]
30034 then
30035     true
30036 else
30037     exit
30038 fi
30039 #
30040 T_ST=‘objdump -h $FILE_ELF | grep -F ".text" | g_st‘
30041 T_VM=‘objdump -h $FILE_ELF | grep -F ".text" | g_vma‘
30042 T_SZ=‘objdump -h $FILE_ELF | grep -F ".text" | g_sz‘
30043 #
30044 R_ST=‘objdump -h $FILE_ELF | grep -F ".rodata" | g_st‘
30045 R_VM=‘objdump -h $FILE_ELF | grep -F ".rodata" | g_vma‘
30046 R_SZ=‘objdump -h $FILE_ELF | grep -F ".rodata" | g_sz‘
30047 #
30048 D_ST=‘objdump -h $FILE_ELF | grep -F ".data" | g_st‘
30049 D_VM=‘objdump -h $FILE_ELF | grep -F ".data" | g_vma‘
30050 D_SZ=‘objdump -h $FILE_ELF | grep -F ".data" | g_sz‘
30051 #
30052 # Convert to decimal
30053 #
30054 T_ST=‘printf "%i" 0x$T_ST‘
30055 T_VM=‘printf "%i" 0x$T_VM‘
30056 T_SZ=‘printf "%i" 0x$T_SZ‘
30057 #
30058 R_ST=‘printf "%i" 0x$R_ST‘
30059 R_VM=‘printf "%i" 0x$R_VM‘
30060 R_SZ=‘printf "%i" 0x$R_SZ‘
30061 #
30062 D_ST=‘printf "%i" 0x$D_ST‘
30063 D_VM=‘printf "%i" 0x$D_VM‘
30064 D_SZ=‘printf "%i" 0x$D_SZ‘
30065
30066
30067 if [ "$R_SZ" = "$D_VM" ]
30068 then
30069     dd if=$FILE_ELF of=$FILE_OS32.text \
30070        bs=1 skip=$T_ST count=$T_SZ 2> "/dev/null"
30071     dd if=$FILE_ELF of=$FILE_OS32.rodata \
30072        bs=1 skip=$R_ST count=$R_SZ 2> "/dev/null"
30073     dd if=$FILE_ELF of=$FILE_OS32.data \
30074        bs=1 skip=$D_ST count=$D_SZ 2> "/dev/null"
30075     cat $FILE_OS32.text $FILE_OS32.rodata  \
30076         $FILE_OS32.data > $FILE_OS32
30077     #
30078     #rm $FILE_OS32.text
30079     #rm $FILE_OS32.rodata
30080     #rm $FILE_OS32.data
30081     #
30082     chmod a+x $FILE_OS32
30083 elif [ "$R_SZ" -lt "$D_VM" ]
30084 then
30085     dd if=$FILE_ELF of=$FILE_OS32.text \
30086        bs=1 skip=$T_ST count=$T_SZ 2> "/dev/null"
30087     dd if=$FILE_ELF of=$FILE_OS32.rodata \
30088        bs=1 skip=$R_ST count=$R_SZ 2> "/dev/null"
30089     dd if=/dev/zero of=$FILE_OS32.rodata-space \
30090        bs=1 count=$(($D_VM-$R_SZ)) 2> "/dev/null"
30091     dd if=$FILE_ELF of=$FILE_OS32.data \
30092        bs=1 skip=$D_ST count=$D_SZ 2> "/dev/null"
30093
30094     cat $FILE_OS32.text \
30095         $FILE_OS32.rodata \
30096         $FILE_OS32.rodata-space \
30097         $FILE_OS32.data > $FILE_OS32
30098     #
30099     #rm $FILE_OS32.text
30100     #rm $FILE_OS32.rodata
30101     #rm $FILE_OS32.rodata-space
30102     #rm $FILE_OS32.data
30103     #
30104     chmod a+x $FILE_OS32
30105 else
30106     echo "[$0] ERROR: $FILE_ELF has DATA section"
30107     echo "[$0]        not contiguous:"
30108     echo "[$0]        RODATA end at $R_SZ, and DATA"
30109     echo "[$0]        should start at $D_VM!"
30110 fi
```

94.1.4 fdisk

«

Si veda la sezione 85.1.

```
40001 #!/bin/sh
40002 #
40003 #
40004 #
```



```
40005 . ./file_image_functions
40006 #
40007 if [ -z "$1" ]
40008 then
40009     echo "$0 DISK_IMAGE_FILE"
40010 else
40011     file_image_fdisk "$1"
40012 fi
40013 #
```

94.1.5 file_image_functions

Si veda la sezione 85.1.

```
50001 #
50002 # file_image_fdisk IMAGE_FILE
50003 #
50004 file_image_fdisk () {
50005   #
50006   local FNAME="$1"
50007   local FSIZE=0
50008   local CYLINDERS=0
50009   local HEADS=0
50010   local SECTORS=63
50011   #
50012   if   [ -z "$FNAME" ]
50013   then
50014       echo "[$0] No file name."
50015       return 1
50016   elif [ ! -r "$FNAME" ]
50017   then
50018       echo "[$0] Cannot read file \"$FNAME\"."
50019       return 1
50020   fi
50021   #
50022   # Get size
50023   #
50024   FSIZE=‘du -k "$FNAME" | sed "s/[[:space:]].*$//"‘
50025   #
50026   # Set geometry
50027   #
50028   if [ "$FSIZE" -le "516096" ]
50029   then
50030       HEADS="16"
50031   else
50032       HEADS="255"
50033   fi
50034   #
50035   CYLINDERS=$(($FSIZE*2/$SECTORS/$HEADS))
50036   #
50037   # Run fdisk.
50038   #
50039   fdisk -C $CYLINDERS -H $HEADS -S $SECTORS $FNAME
50040 }
50041 #
50042 # file_image_partition_start IMAGE_FILE PART_NUMBER
50043 #
50044 file_image_partition_start () {
50045   #
50046   local FNAME="$1"
50047   local PART_NUMBER="$2"
50048   local PART_START=0
50049   #
50050   # Get partition start
50051   #
50052   PART_START=‘sfdisk -d $FNAME \
50053     | grep -F "$FNAME$PART_NUMBER" \
50054     | sed "s/^.*start=[[:space:]]*//" | sed "s/,.*$//"‘
50055   #
50056   echo "$PART_START"
50057 }
50058 #
50059 # file_image_partition_size IMAGE_FILE PART_NUMBER
50060 #
50061 file_image_partition_size () {
50062   #
50063   local FNAME="$1"
50064   local PART_NUMBER="$2"
50065   local PART_SIZE=0
50066   #
50067   # Get partition start
50068   #
50069   PART_SIZE=‘sfdisk -d $FNAME \
50070     | grep -F "$FNAME$PART_NUMBER" \
50071     | sed "s/^.*size=[[:space:]]*//" | sed "s/,.*$//"‘
50072   #
50073   echo "$PART_SIZE"
```

```
50074 }
50075 #
50076 # file_image_partition_id IMAGE_FILE PART_NUMBER
50077 #
50078 file_image_partition_id () {
50079   #
50080   local FNAME="$1"
50081   local PART_NUMBER="$2"
50082   local PART_ID=0
50083   #
50084   # Get partition start
50085   #
50086   PART_ID=‘sfdisk -d $FNAME \
50087    | grep -F "$FNAME$PART_NUMBER" \
50088    | sed "s/^.*Id=[[:space:]]*//" | sed "s/,.*$//"‘
50089   #
50090   echo "$PART_ID"
50091 }
50092 #
50093 # file_image_partition_format IMAGE_FILE PART_NUMBER \
50094 #                             [dos|minix]
50095 #
50096 file_image_partition_format () {
50097   #
50098   local FNAME="$1"
50099   local PART_NUMBER="$2"
50100   local PART_FORMAT="$3"
50101   local PART_START=‘file_image_partition_start \
50102                     $FNAME $PART_NUMBER‘
50103   local PART_SIZE=‘file_image_partition_size $FNAME \
50104                    $PART_NUMBER‘
50105   local PART_ID=‘file_image_partition_id $FNAME \
50106                  $PART_NUMBER‘
50107   #
50108   #
50109   #
50110   if [ "$PART_SIZE" -eq "0" ]
50111   then
50112       exit
50113   fi
50114   #
50115   # Get partition into a file.
50116   #
50117   dd if="$FNAME" \
50118      of="$FNAME.part$PART_NUMBER.tmp" \
50119      bs=512 \
50120      skip="$PART_START" \
50121      count="$PART_SIZE"
50122   #
50123   # Format.
50124   #
50125   if [ "$PART_FORMAT" = "dos" ] \
50126     || [ "$PART_FORMAT" = "msdos" ]
50127   then
50128       mkfs.msdos -v "$FNAME.part$PART_NUMBER.tmp"
50129   else
50130       mkfs.minix -n 14 "$FNAME.part$PART_NUMBER.tmp"
50131   fi
50132   #
50133   # Put formatted partition into the file.
50134   #
50135   dd if="$FNAME.part$PART_NUMBER.tmp" \
50136      of="$FNAME" \
50137      bs=512 \
50138      seek="$PART_START" \
50139      count="$PART_SIZE" \
50140      conv=notrunc
50141   #
50142   # Remove temporary file.
50143   #
50144   rm "$FNAME.part$PART_NUMBER.tmp"
50145   #
50146 }
50147 #
50148 # file_image_partition_mount IMAGE_FILE PART_NUMBER
50149 #
50150 file_image_partition_mount () {
50151   #
50152   local FNAME="$1"
50153   local PART_NUMBER="$2"
50154   local PART_START=‘file_image_partition_start \
50155                     $FNAME $PART_NUMBER‘
50156   local PART_SIZE=‘file_image_partition_size \
50157                    $FNAME $PART_NUMBER‘
50158   local PART_ID=‘file_image_partition_id \
50159                  $FNAME $PART_NUMBER‘
50160   local MOUNT_POINT
```



```
50161   #
50162   #
50163   #
50164   if [ "$PART_SIZE" -eq "0" ]
50165   then
50166       exit
50167   fi
50168   #
50169   # Find mount point.
50170   #
50171   MOUNT_POINT=‘basename $FNAME‘
50172   MOUNT_POINT="/mnt/${MOUNT_POINT}.$PART_NUMBER"
50173   #
50174   #
50175   #
50176   sync
50177   #
50178   umount "$MOUNT_POINT" 2> "/dev/null"
50179   umount "$MOUNT_POINT" 2> "/dev/null"
50180   umount "$MOUNT_POINT" 2> "/dev/null"
50181   #
50182   mkdir -p "$MOUNT_POINT" 2> "/dev/null"
50183   #
50184   # Get partition into a file.
50185   #
50186   dd if="$FNAME" \
50187      of="$FNAME.part$PART_NUMBER.tmp" \
50188      bs=512 \
50189      skip="$PART_START" \
50190      count="$PART_SIZE"
50191   #
50192   # Check partition.
50193   #
50194   fsck -f -v -r "$FNAME.part$PART_NUMBER.tmp"
50195   #
50196   # Put formatted partition into the file.
50197   #
50198   dd if="$FNAME.part$PART_NUMBER.tmp" \
50199      of="$FNAME" \
50200      bs=512 \
50201      seek="$PART_START" \
50202      count="$PART_SIZE" \
50203      conv=notrunc
50204   #
50205   # Remove temporary file.
50206   #
50207   rm "$FNAME.part$PART_NUMBER.tmp"
50208   #
50209   # Mount the partition.
50210   #
50211   mount -o loop,offset=$(($PART_START*512)) \
50212         -t auto "$FNAME" "$MOUNT_POINT"
50213   #
50214 }
50215 #
50216 # file_image_partition_syslinux IMAGE_FILE PART_NUMBER
50217 #
50218 file_image_partition_syslinux () {
50219   #
50220   local FNAME="$1"
50221   local PART_NUMBER="$2"
50222   local PART_START=‘file_image_partition_start \
50223                     $FNAME $PART_NUMBER‘
50224   local PART_SIZE=‘file_image_partition_size \
50225                    $FNAME $PART_NUMBER‘
50226   local PART_ID=‘file_image_partition_id \
50227                  $FNAME $PART_NUMBER‘
50228   #
50229   #
50230   #
50231   if [ "$PART_SIZE" -eq "0" ]
50232   then
50233       exit
50234   fi
50235   #
50236   # Get partition into a file.
50237   #
50238   dd if="$FNAME" \
50239      of="$FNAME.part$PART_NUMBER.tmp" \
50240      bs=512 \
50241      skip="$PART_START" \
50242      count="$PART_SIZE"
50243   #
50244   # Syslinux
50245   #
50246   syslinux "$FNAME.part$PART_NUMBER.tmp"
50247   #
```

```
50248    # Put altered partition into the file.
50249    #
50250    dd if="$FNAME.part$PART_NUMBER.tmp" \
50251       of="$FNAME" \
50252       bs=512 \
50253       seek="$PART_START" \
50254       count="$PART_SIZE" \
50255       conv=notrunc
50256    #
50257    # Remove temporary file.
50258    #
50259    rm "$FNAME.part$PART_NUMBER.tmp"
50260    #
50261    # Fix MBR
50262    #
50263    install-mbr $FNAME
50264  }
50265
```

94.1.6 format

«

Si veda la sezione 85.1.

```
60001  #!/bin/sh
60002  #
60003  #
60004  #
60005  . ./file_image_functions
60006  #
60007  if [ -z "$1" ] \
60008    || [ -z "$2" ] \
60009    || [ "$2" -lt "1" ] \
60010    || [ "$2" -gt "4" ]
60011  then
60012      echo "Usage:"
60013      echo ""
60014      echo "$0 DISK_IMAGE_FILE PART_NUMBER dos|minix"
60015      echo ""
60016      echo "The partition number must be between"
60017      echo " 1 and 4. No extended partitions are"
60018      echo "handled!"
60019  else
60020      file_image_partition_format "$1" "$2" "$3"
60021  fi
60022  #
```

94.1.7 kernel.ld

«

Si veda la sezione 84.2.2.

```
70001  /*****************************************************
70002   * The code will start at address 0x100000, that is at
70003   * 1 Mibyte, because it is the place where GRUB will
70004   * place it.
70005   *
70006   * The kernel is divided into 'TEXT' (code), 'DATA' and
70007   * 'BSS'.
70008   * Between the TEXT and the DATA there is a gap to
70009   * align the data at 4 Kibyte boundary (0x1000), to
70010   * allow memory management for it.
70011   *
70012   * The stack will be placed at the beginning of the
70013   * BSS.
70014   *
70015   * The kernel starts with file 'kernel/main/crt0.s',
70016   * at the label 'startup'.
70017   *****************************************************/
70018  ENTRY (kstartup)
70019  SECTIONS {
70020      . = 0x00100000;
70021      _k_start = .;
70022      _k_text_start = .;
70023      .text : {
70024          *(.text)
70025      }
70026      _k_text_end = .;
70027      . = ALIGN (0x1000);
70028      _k_data_start = .;
70029      .rodata : {
70030          *(.rodata)
70031      }
70032      . = ALIGN (0x4);
70033      .data : {
70034          *(.data)
70035      }
70036      _k_data_end = .;
70037      . = ALIGN (0x4);
```



```
70038      _k_bss_start = .;
70039      .bss : {
70040          *(COMMON)
70041          *(.bss)
70042      }
70043      _k_bss_end = .;
70044      _k_end = .;
70045  }
```

94.1.8 makeit.sep

«

Si veda la sezione 91.3.

```
80001  #!/bin/sh
80002  #
80003  # makeit... separated: text and data have separate
80004  # segments.
80005  #
80006  OPTION="$1"
80007  OS32PATH=""
80008  #
80009  #
80010  #
80011  edition () {
80012    local EDITION="kernel/main/build.h"
80013    echo -n                                > $EDITION
80014    echo -n "#define BUILD_DATE \""      >> $EDITION
80015    echo -n `date "+%Y%m%d%H%M"`         >> $EDITION
80016    echo  "\""  >> $EDITION
80017  }
80018  #
80019  #
80020  #
80021  makefile () {
80022    #
80023    local MAKEFILE="Makefile"
80024    local TAB=`printf "\t"`
80025    #
80026    local SOURCE_C=""
80027    local C=""
80028    local SOURCE_S=""
80029    local S=""
80030    #
80031    local c
80032    local s
80033    #
80034    # Find C source files.
80035    #
80036    for c in *.c
80037    do
80038        if [ -f $c ]
80039        then
80040        C=`basename $c .c`
80041            SOURCE_C="$SOURCE_C $C"
80042        fi
80043    done
80044    #
80045    # Find ASM source files.
80046    #
80047    for s in *.s
80048    do
80049        if [ -f $s ]
80050        then
80051        S=`basename $s .s`
80052            SOURCE_S="$SOURCE_S $S"
80053        fi
80054    done
80055    #
80056    # Prepare the Makefile. Option '-g' is for debugging
80057    # symbols.
80058    #
80059    echo -n                                 > $MAKEFILE
80060    echo "# This file was made "            >> $MAKEFILE
80061    echo "# automatically"                  >> $MAKEFILE
80062    echo "# by the script \`makeit', based" >> $MAKEFILE
80063    echo "# on the directory content."      >> $MAKEFILE
80064    echo "# Please use \`makeit' to "       >> $MAKEFILE
80065    echo "# compile and"                    >> $MAKEFILE
80066    echo "# \`makeit clean\' to clean "     >> $MAKEFILE
80067    echo "# directories."                   >> $MAKEFILE
80068    echo "#"                              >> $MAKEFILE
80069    echo "#"                              >> $MAKEFILE
80070    echo "c = $SOURCE_C"                  >> $MAKEFILE
80071    echo "#"                              >> $MAKEFILE
80072    echo "s = $SOURCE_S"                  >> $MAKEFILE
80073    echo "#"                              >> $MAKEFILE
80074    echo "all: \$(s) \$(c)"               >> $MAKEFILE
```

```
80075   echo "#"                                  >> $MAKEFILE
80076   echo "clean:"                              >> $MAKEFILE
80077   echo "${TAB}@rm *~ *.o *.ELF *.text " \
80078        "*.rodata *.rodata-space " \
80079        "*.data \$(c) 2> /dev/null ; pwd"    >> $MAKEFILE
80080   echo "#"                                  >> $MAKEFILE
80081   echo "\$(s):"                              >> $MAKEFILE
80082   echo "${TAB}@echo \$@.s"                  >> $MAKEFILE
80083   echo "${TAB}@as -o \$@.o \$@.s"           >> $MAKEFILE
80084   echo "#"                                  >> $MAKEFILE
80085   echo "\$(c):"                              >> $MAKEFILE
80086   echo "${TAB}@echo \$@.c"                  >> $MAKEFILE
80087   echo "${TAB}@gcc -O0 -Wall -Werror "    \
80088        "-Wno-unused-but-set-variable"     \
80089        "-g"                               \
80090        "-o \$@.o -c \$@.c"                \
80091        "-nostdinc -nostdlib "             \
80092        "-nostartfiles -nodefaultlibs" \
80093        "-I " \
80094        "-I. " \
80095        "-I$OS32PATH/lib " \
80096        "-I$OS32PATH/ " \
80097        "-I../include -I../../include " \
80098        "-I../../../include"  >> $MAKEFILE
80099   #
80100 }
80101 #
80102 #
80103 #
80104 main () {
80105   #
80106   local CURDIR=‘pwd‘
80107   local OBJECTS
80108   local d
80109   local c
80110   local s
80111   local o
80112   #
80113   edition
80114   #
80115   # Copia dello scheletro
80116   #
80117   if [ "$OPTION" = "clean" ]
80118   then
80119       #
80120       # La copia non va fatta.
80121       #
80122       true
80123   else
80124       cp -dpRv skel/etc   /mnt/disk.hda.2/
80125       cp -dpRv skel/dev   /mnt/disk.hda.2/
80126       mkdir               /mnt/disk.hda.2/mnt/
80127       mkdir               /mnt/disk.hda.2/tmp/
80128       chmod 0777          /mnt/disk.hda.2/tmp/
80129       mkdir               /mnt/disk.hda.2/usr/
80130       mkdir               /mnt/disk.hda.2/var/
80131       cp -dpRv skel/root  /mnt/disk.hda.2/
80132       cp -dpRv skel/home  /mnt/disk.hda.2/
80133       cp -dpRv skel/usr/* /mnt/disk.hda.2/usr/
80134       cp -dpRv skel/var/* /mnt/disk.hda.2/var/
80135   fi
80136   #
80137   for d in ‘find .‘
80138   do
80139     if [ -d "$d" ]
80140     then
80141       #
80142       # Are there C or ASM source files?
80143       #
80144       c=‘echo $d/*.c | sed "s/ .*//"‘
80145       s=‘echo $d/*.s | sed "s/ .*//"‘
80146       #
80147       if [ -f "$c" ] || [ -f "$s" ]
80148       then
80149           CURDIR=‘pwd‘
80150           cd $d
80151           #
80152           # Build the new makefile
80153           #
80154           makefile
80155           #
80156           # Clean the directory
80157           #
80158           make clean
80159           #
80160           #
80161           #
```



```
80162           if [ "$OPTION" = "clean" ]
80163           then
80164               #
80165               # Nothing else to do: the clean was
80166               # just made.
80167               #
80168               true
80169           else
80170               if ! make
80171               then
80172                   cd "$CURDIR"
80173                   exit
80174               fi
80175           fi
80176           cd "$CURDIR"
80177       fi
80178     fi
80179   done
80180   #
80181   cd "$CURDIR"
80182   #
80183   #
80184   #
80185   if [ "$OPTION" = "clean" ]
80186   then
80187     true
80188   else
80189     #
80190     echo "Link kernel"
80191     #
80192     OBJECTS=""
80193     #
80194     for o in ‘find . -name \*.o -print‘
80195     do
80196       if    [ "$o" = "./kernel/main/crt0.o" ] \
80197         || [ "$o" = "./kernel/main/kmain.o" ] \
80198         || [ "$o" = "./kernel/main/stack.o" ] \
80199         || [ ! -e "$o" ] \
80200         || echo "$o" | grep -F "./applic/" \
80201             > "/dev/null" \
80202         || echo "$o" | grep -F "./ported/" \
80203             > "/dev/null"
80204       then
80205         true
80206       else
80207         OBJECTS="$OBJECTS $o"
80208       fi
80209     done
80210     #
80211     # The kernel must be ELF, because Grub will not
80212     # recognize it otherwise.
80213     #
80214     ld  --script=kernel.ld \
80215         --oformat elf32-i386 \
80216         -o kimage \
80217         ./kernel/main/crt0.o \
80218         $OBJECTS \
80219         ./kernel/main/kmain.o \
80220         ./kernel/main/stack.o
80221     #
80222     cp -f kimage /mnt/disk.hda.1/kimage
80223     sync
80224     #
80225     # Collegamento delle applicazioni di os32.
80226     #
80227     OBJLIB=""
80228     #
80229     for o in ‘find lib      -name \*.o -print‘
80230     do
80231         OBJLIB="$OBJLIB $o"
80232     done
80233     #
80234     echo "Link applic"
80235     #
80236     # Scansione delle applicazioni interne.
80237     #
80238     for o in ‘find applic     -name \*.o -print‘
80239     do
80240       if    [ "$o" = "applic/crt0.o" ] \
80241         || [ ! -e "$o" ] \
80242         || echo "$o" | grep ".crt0.o$" > /dev/null \
80243         || echo "$o" | grep ".crt0.mer.o$" \
80244             > /dev/null \
80245         || echo "$o" | grep ".crt0.sep.o$" > /dev/null
80246       then
80247         #
80248         # Il file non esiste oppure si tratta di
```

```
80249         # '...crt0.s'.
80250         #
80251         true
80252       else
80253         #
80254         # File oggetto differente da '...crt0.s'.
80255         #
80256         EXEC='echo "$o" | sed "s/\.o$//"'
80257         BASENAME='basename $o .o'
80258         if [ -e "applic/$BASENAME.crt0.sep.o" ]
80259         then
80260           #
80261           # Qui c'è un file '...crt0.o' specifico.
80262           #
80263           rm $EXEC $EXEC.ELF  2> "/dev/null"
80264           ld  --no-check-sections \
80265               --oformat elf32-i386 \
80266               --script=applic.sep.ld \
80267               -o $EXEC.ELF \
80268               ./applic/$BASENAME.crt0.sep.o \
80269               $o \
80270               $OBJLIB
80271               #
80272               ./elf-to-os32 $EXEC.ELF $EXEC
80273         else
80274           #
80275           # Qui si usa il file 'crt0.sep.o' generale.
80276           #
80277           rm $EXEC $EXEC.ELF  2> "/dev/null"
80278           ld  --script=applic.sep.ld \
80279               --no-check-sections \
80280               --oformat elf32-i386 \
80281               -o $EXEC.ELF \
80282               ./applic/crt0.sep.o \
80283               $o \
80284               $OBJLIB
80285               #
80286               ./elf-to-os32 $EXEC.ELF $EXEC
80287         fi
80288         #
80289         if [ -x "applic/$BASENAME" ]
80290         then
80291           if mount | grep /mnt/disk.hda.2 > /dev/null
80292           then
80293             mkdir /mnt/disk.hda.2/bin/ 2> /dev/null
80294             rm /mnt/disk.hda.2/bin/$EXEC 2> "/dev/null"
80295             cp -f "$EXEC" /mnt/disk.hda.2/bin
80296           else
80297             echo "[$0] Cannot copy the application"
80298             echo "[$0]   $BASENAME inside the disk"
80299             echo "[$0]   image!"
80300             break
80301           fi
80302         fi
80303       fi
80304     done
80305     sync
80306     #
80307     echo "Link ported"
80308     #
80309     # Scansione delle applicazioni adattate.
80310     #
80311     for a in ported/*
80312     do
80313       if [ -d $a ]
80314       then
80315         OBJECTS=""
80316         for o in 'find $a  -name \*.o -print'
80317         do
80318           if    [ "$o" = "$a/crt0.o" ] \
80319             || [ ! -e "$o" ] \
80320             || echo "$o" | grep "crt0.o$" > /dev/null \
80321             || echo "$o" | grep "crt0.mer.o$" \
80322                 > /dev/null \
80323             || echo "$o" | grep "crt0.sep.o$" \
80324                 > /dev/null
80325           then
80326             #
80327             # Il file non esiste oppure si tratta di
80328             # '...crt0.s'.
80329             #
80330             true
80331           else
80332             OBJECTS="$OBJECTS $o"
80333           fi
80334         done
80335         #
```



```
80336         # File oggetto differente da '...crt0.s'.
80337         #
80338         BASENAME='basename $a'
80339         EXEC="$a/$BASENAME"
80340         #
80341         rm $EXEC $EXEC.ELF  2> "/dev/null"
80342         #
80343         echo ld  --script=applic.sep.ld \
80344             --no-check-sections \
80345             --oformat elf32-i386 \
80346             -o $EXEC.ELF \
80347             $a/crt0.sep.o \
80348             $OBJECTS \
80349             $OBJLIB > link-$BASENAME.link
80350             #
80351         echo ./elf-to-os32 $EXEC.ELF $EXEC \
80352              >> link-$BASENAME.link
80353         #
80354         ld  --script=applic.sep.ld \
80355             --no-check-sections \
80356             --oformat elf32-i386 \
80357             -o $EXEC.ELF \
80358             $a/crt0.sep.o \
80359             $OBJECTS \
80360             $OBJLIB
80361             #
80362         ./elf-to-os32 $EXEC.ELF $EXEC
80363         #
80364         if [ -x "$EXEC" ]
80365         then
80366           if mount | grep /mnt/disk.hda.2 > /dev/null
80367           then
80368             mkdir /mnt/disk.hda.2/bin/ 2> /dev/null
80369             rm /mnt/disk.hda.2/bin/$EXEC 2> "/dev/null"
80370             cp -f "$EXEC" /mnt/disk.hda.2/bin
80371           else
80372             echo "[$0] Cannot copy the application "
80373             echo "[$0]   $BASENAME inside the disk "
80374             echo "[$0]   image!"
80375             break
80376           fi
80377         fi
80378       fi
80379     done
80380     sync
80381   fi
80382 }
80383 #
80384 # Start.
80385 #
80386 if [ -d kernel ]   && \
80387    [ -d lib  ]
80388 then
80389     OS32PATH='pwd'
80390     main
80391 else
80392     echo "[$0] Running from a wrong directory!"
80393 fi
```

94.1.9 qemu

Si veda la sezione 85.4.

```
90001 #!/bin/sh
90002 #
90003 #
90004 #
90005 if [ -n "$DISPLAY" ]
90006 then
90007     CURSES=""
90008 else
90009     CURSES="-curses"
90010 fi
90011
90012 if [ "$EUID" = "0" ]
90013 then
90014   #   172.21.11.18                    172.21.11.16
90015   #  >-----------point to point -----> >-------os32
90016   #   tap0 (linux)                    net1
90017   #
90018   # Dal lato Linux:
90019   #   ifconfig tap0 172.21.11.18 pointopoint \
90020   #             172.21.11.16 netmask 255.255.255.255
90021   #   route add -host 172.21.11.16 gw 172.21.11.18
90022   #
90023   # Dalla macchina 172.21.254.254:
90024   #   route add -host 172.21.11.16 gw 172.21.11.18
```

```
90025   #
90026   qemu $CURSES \
90027     -hda disk.hda \
90028     -net nic,macaddr=b0:c4:20:00:00:00,model=ne2k_pci \
90029     -net tap,ifname=tap0,script=./tap0 \
90030     -boot c
90031 else
90032   echo "[$0] Qemu avviato senza privilegi: non"
90033   echo "[$0] funziona la rete!"
90034   echo "[$0] Premi Invio per continuare"
90035   read
90036
90037   qemu $CURSES \
90038     -hda disk.hda \
90039     -net nic,macaddr=b0:c4:20:00:00:00,model=ne2k_pci \
90040     -net user,net=172.21.0.0/16,host=172.21.254.254,\
90041 restrict=n \
90042     -boot c
90043 fi
90044
```

94.1.10 syslinux

«

Si veda la sezione 85.1.

```
100001 #!/bin/sh
100002 #
100003 #
100004 #
100005 . ./file_image_functions
100006 #
100007 if [ -z "$1" ] || [ -z "$2" ] || [ "$2" -lt "1" ] \
100008   || [ "$2" -gt "4" ]
100009 then
100010     echo "Usage:"
100011     echo ""
100012     echo "$0 DISK_IMAGE_FILE PART_NUMBER"
100013     echo ""
100014     echo "The partition number must be between"
100015     echo " 1 and 4."
100016     echo "No extended partitions are handled!"
100017 else
100018     file_image_partition_syslinux "$1" "$2"
100019 fi
100020 #
```

94.1.11 tap0

«

Si veda la sezione 85.4.

```
110001 #!/bin/sh
110002
110003
110004 ifconfig tap0 172.21.11.18 pointopoint 172.21.11.16 \
110005               netmask 255.255.255.255
110006 route add -host 172.21.11.16 gw 172.21.11.18
110007
110008
```

94.2 os32: «kernel/blk.h»

«

Si veda la sezione 93.3.

```
120001 #ifndef _KERNEL_BLK_H
120002 #define _KERNEL_BLK_H 1
120003 //-------------------------------------------------------
120004 #include <sys/types.h>
120005 #include <kernel/driver/ata.h>
120006 //-------------------------------------------------------
120007 #define BLK_SIZE          ATA_SECTOR_SIZE       // [1]
120008 //
120009 // [1] This value should be the same as the mass memory
120010 //     sector size, or a multiple. But with a multiple,
120011 //     all simple read and write will be doubled, and
120012 //     some race will lock the system, because read and
120013 //     write are too frequent for the poor ATA driver
120014 //     that I have. :-(
120015 //
120016 //-------------------------------------------------------
120017 #define BLK_CACHE_SIZE          128     // This is
120018                                         // free!
120019 #define BLK_CACHE_MAX_AGE       BLK_CACHE_SIZE-1
120020 typedef struct
120021 {
120022   unsigned int age;     // 0=last used
120023   // DEV_CACHE_MAX_AGE=older
120024   dev_t device;
```



```
120025   unsigned int n;
120026   char block[BLK_SIZE];
120027 } blk_cache_t;
120028
120029 extern blk_cache_t blk_table[BLK_CACHE_SIZE];
120030 //-------------------------------------------------------
120031 void *blk_ata (dev_t device, int rw, unsigned int n,
120032               void *buffer);
120033 //-------------------------------------------------------
120034 void blk_cache_init (void);
120035 void blk_cache_check (void);
120036 void *blk_cache_read (dev_t device, unsigned int n);
120037 void *blk_cache_save (dev_t device, unsigned int n,
120038                       void *block);
120039 //-------------------------------------------------------
120040
120041 #endif
```


94.2.1 kernel/blk/blk_ata.c

«

Si veda la sezione 93.3.1.

```
130001 #include <sys/os32.h>
130002 #include <kernel/blk.h>
130003 #include <kernel/dev.h>
130004 #include <kernel/driver/ata.h>
130005 #include <kernel/lib_k.h>
130006 #include <kernel/dm.h>
130007 #include <sys/types.h>
130008 #include <ctype.h>
130009 #include <string.h>
130010 //-------------------------------------------------------
130011 #define DEBUG 0
130012 //-------------------------------------------------------
130013 void *
130014 blk_ata (dev_t device, int rw, unsigned int n, void *buffer)
130015 {
130016   ata_sector_t *destination = buffer;
130017   ata_sector_t *source = buffer;
130018   int dev_minor = minor (device);
130019   int d = ((dev_minor & 0x00F0) >> 4);
130020   ptrdiff_t ptrdiff;
130021   int drive;
130022   unsigned int sector = n * (BLK_SIZE / ATA_SECTOR_SIZE);
130023   size_t count = (BLK_SIZE / ATA_SECTOR_SIZE);
130024   int status;
130025   int c;
130026   void *cache;
130027   //
130028   // Convert the table pointer to a drive number.
130029   //
130030   ptrdiff = (((intptr_t) dm_table[d].table)
130031              - ((intptr_t) ata_table));
130032   drive = ptrdiff / (sizeof (ata_t));
130033   //
130034   if (DEBUG)
130035     {
130036       k_printf ("%s: R/W=%i dev=%04x n=%ui ", __FILE__,
130037                 rw, (int) device, n);
130038       blk_cache_check ();
130039     }
130040   //
130041   // If reading, check if we already have inside
130042   // cache.
130043   //
130044   if (rw == DEV_READ)
130045     {
130046       cache = blk_cache_read (device, n);
130047       if (cache != NULL)
130048         {
130049           return (cache);
130050         }
130051     }
130052   //
130053   // Read or write.
130054   //
```

```
130055    for (c = 0, status = 0;
130056         c < count && status == 0; c++, sector++)
130057      {
130058        //
130059        if (DEBUG)
130060          {
130061            k_printf ("sec=%i ", sector);
130062          }
130063        //
130064        if (rw == DEV_READ)
130065          {
130066            status = ata_read_sector (drive, sector,
130067                                      &destination[c]);
130068          }
130069        else
130070          {
130071            status = ata_write_sector (drive, sector,
130072                                       &source[c]);
130073          }
130074      }
130075    //
130076    // If a block was read or written inside ATA
130077    // hardware, then
130078    // save it inside the cache.
130079    //
130080    if (status == 0)
130081      {
130082        cache = blk_cache_save (device, n, buffer);
130083      }
130084    else
130085      {
130086        cache = NULL;
130087      }
130088    //
130089    //
130090    //
130091    if (DEBUG)
130092      {
130093        k_printf ("\n");
130094      }
130095    //
130096    return (cache);
130097  }
```

94.2.2 kernel/blk/blk_cache_check.c

«

Si veda la sezione 93.3.2.

```
140001  #include <kernel/blk.h>
140002  #include <string.h>
140003  #include <kernel/lib_k.h>
140004  //-----------------------------------------------------------
140005  void
140006  blk_cache_check (void)
140007  {
140008    int i;
140009    int j;
140010    //
140011    // check if all ages are present.
140012    //
140013    for (i = 0; i < BLK_CACHE_SIZE; i++)
140014      {
140015        if (blk_table[i].age > BLK_CACHE_MAX_AGE)
140016          {
140017            k_printf
140018              ("blk_table[%i].age > BLK_CACHE_MAX_AGE\n", i);
140019            return;
140020          }
140021        for (j = 0; j < BLK_CACHE_SIZE; j++)
140022          {
140023            if (j != i
140024                && blk_table[i].age == blk_table[j].age)
140025              {
140026                k_printf
140027                  ("blk_table[%i].age == "
140028                   "blk_table[%i].age\n", i, j);
140029                return;
140030              }
140031          }
140032      }
140033  }
```


94.2.3 kernel/blk/blk_cache_init.c

«

Si veda la sezione 93.3.3.

```
150001  #include <kernel/blk.h>
150002  //-----------------------------------------------------------
150003  void
150004  blk_cache_init (void)
150005  {
150006    int i;
150007    for (i = 0; i < BLK_CACHE_SIZE; i++)
150008      {
150009        blk_table[i].age = i;     // Age is from 0 to
150010        // BLK_CACHE_MAX_AGE.
150011        blk_table[i].device = 0;
150012        blk_table[i].n = 0;
150013      }
150014  }
```

94.2.4 kernel/blk/blk_cache_read.c

«

Si veda la sezione 93.3.4.

```
160001  #include <kernel/blk.h>
160002  #include <string.h>
160003  //-----------------------------------------------------------
160004  void *
160005  blk_cache_read (dev_t device, unsigned int n)
160006  {
160007    int i;
160008    int j;
160009    int age;
160010    //
160011    device &= 0xFFF0;
160012    //
160013    for (i = 0; i < BLK_CACHE_SIZE; i++)
160014      {
160015        if (blk_table[i].device == device
160016            && blk_table[i].n == n)
160017          {
160018            age = blk_table[i].age;
160019            for (j = 0; j < BLK_CACHE_SIZE; j++)
160020              {
160021                if (blk_table[j].age < age)
160022                  {
160023                    blk_table[j].age++;
160024                  }
160025              }
160026            blk_table[i].age = 0;
160027            //
160028            return (&blk_table[i].block);
160029          }
160030      }
160031    return (NULL);
160032  }
```

94.2.5 kernel/blk/blk_cache_save.c

«

Si veda la sezione 93.3.4.

```
170001  #include <kernel/blk.h>
170002  #include <string.h>
170003  //-----------------------------------------------------------
170004  void *
170005  blk_cache_save (dev_t device, unsigned int n, void *block)
170006  {
170007    int i;
170008    int j;
170009    int age;
170010    //
170011    device &= 0xFFF0;
170012    //
170013    // Look inside the cache, if we already have
170014    // that old block.
170015    //
170016    for (i = 0; i < BLK_CACHE_SIZE; i++)
170017      {
170018        if (blk_table[i].device == device
170019            && blk_table[i].n == n)
170020          {
170021            age = blk_table[i].age;
170022            for (j = 0; j < BLK_CACHE_SIZE; j++)
170023              {
170024                if (blk_table[j].age < age)
170025                  {
170026                    blk_table[j].age++;
170027                  }
170028              }
```

```
170029            blk_table[i].age = 0;
170030            //
170031            // Check if the block is the same memory.
170032            //
170033            if (blk_table[i].block == block)
170034              {
170035                //
170036                // No need to transfer data.
170037                //
170038                ;
170039              }
170040            else
170041              {
170042                memcpy (blk_table[i].block, block,
170043                        (size_t) BLK_SIZE);
170044              }
170045            return (&blk_table[i].block);
170046          }
170047      }
170048    //
170049    // The block is new for the cache: must find
170050    // the older and replace it.
170051    //
170052    for (i = 0; i < BLK_CACHE_SIZE; i++)
170053      {
170054        if (blk_table[i].age == BLK_CACHE_MAX_AGE)
170055          {
170056            for (j = 0; j < BLK_CACHE_SIZE; j++)
170057              {
170058                if (blk_table[j].age < BLK_CACHE_MAX_AGE)
170059                  {
170060                    blk_table[j].age++;
170061                  }
170062              }
170063            blk_table[i].age = 0;
170064            blk_table[i].device = device;
170065            blk_table[i].n = n;
170066            memcpy (blk_table[i].block, block,
170067                    (size_t) BLK_SIZE);
170068            //
170069            return (&blk_table[i].block);
170070          }
170071      }
170072    //
170073    // It should never happen to fail.
170074    //
170075    return (NULL);
170076  }
```

94.2.6 kernel/blk/blk_public.c

«

Si veda la sezione 93.3.

```
180001  #include <kernel/blk.h>
180002  //-------------------------------------------------------
180003  blk_cache_t blk_table[BLK_CACHE_SIZE];
```

94.3 os32: «kernel/dev.h»

«

Si veda la sezione 93.4.

```
190001  #ifndef _KERNEL_DEV_H
190002  #define _KERNEL_DEV_H 1
190003  //-------------------------------------------------------
190004  #include <sys/os32.h>
190005  #include <sys/types.h>
190006  #include <kernel/driver/ata.h>
190007  //-------------------------------------------------------
190008  #define DEV_READ                0
190009  #define DEV_WRITE               1
190010  ssize_t dev_io (pid_t pid, dev_t device, int rw,
190011                  off_t offset, void *buffer,
190012                  size_t size, int *eof);
190013  //-------------------------------------------------------
190014  // The following functions are used only by 'dev_io()'.
190015  //-------------------------------------------------------
190016  ssize_t dev_dm (pid_t pid, dev_t device, int rw,
190017                  off_t offset, void *buffer,
190018                  size_t size, int *eof);
190019  ssize_t dev_ata (pid_t pid, dev_t device, int rw,
190020                   off_t offset, void *buffer,
190021                   size_t size, int *eof);
190022  ssize_t dev_mem (pid_t pid, dev_t device, int rw,
190023                   off_t offset, void *buffer,
190024                   size_t size, int *eof);
190025  ssize_t dev_tty (pid_t pid, dev_t device, int rw,
190026                   off_t offset, void *buffer,
```



```
190027                   size_t size, int *eof);
190028  ssize_t dev_kmem (pid_t pid, dev_t device, int rw,
190029                    off_t offset, void *buffer,
190030                    size_t size, int *eof);
190031  //-------------------------------------------------------
190032  #endif
```


94.3.1 kernel/dev/dev_ata.c

«

Si veda la sezione 93.4.3.

```
200001  #include <sys/os32.h>
200002  #include <kernel/dev.h>
200003  #include <kernel/blk.h>
200004  #include <kernel/driver/ata.h>
200005  #include <kernel/lib_k.h>
200006  #include <kernel/dm.h>
200007  #include <sys/types.h>
200008  #include <ctype.h>
200009  #include <errno.h>
200010  //-------------------------------------------------------
200011  ssize_t
200012  dev_ata (pid_t pid, dev_t device, int rw, off_t offset,
200013           void *buffer, size_t size, int *eof)
200014  {
200015    ssize_t m;
200016    ssize_t n = 0;
200017    unsigned char *data_buffer = buffer;
200018    unsigned int n_blk;
200019    int i;
200020    int j = 0;
200021    char *block;
200022    //
200023    // Read the first block.
200024    //
200025    n_blk = offset / BLK_SIZE;
200026    i = offset % BLK_SIZE;
200027    //
200028    block = blk_ata (device, DEV_READ, n_blk, NULL);
200029    if (block == NULL)
200030      {
200031        errset (errno);
200032        return ((ssize_t) - 1);
200033      }
200034    //
200035    // Read or write.
200036    //
200037    if (rw == DEV_READ)
200038      {
200039        while (size)
200040          {
200041            for (;
200042                 i < BLK_SIZE && size > 0;
200043                 i++, j++, n++, size--, offset++)
200044              {
200045                data_buffer[j] = block[i];
200046              }
200047            if (size)
200048              {
200049                n_blk = offset / BLK_SIZE;
200050                i = offset % BLK_SIZE;
200051                block = blk_ata (device, DEV_READ, n_blk,
200052                                 NULL);
200053                if (block == NULL)
200054                  {
200055                    errset (errno);
200056                    return (n);   // Up to the last
200057                    // block read.
200058                  }
200059              }
200060          }
200061      }
200062    else
200063      {
200064        //
200065        // Write.
```

```
200066        //
200067        while (size)
200068          {
200069            //
200070            // The last block was written, so update
200071            // the counter 'm'.
200072            //
200073            m = n;
200074            //
200075            for (;
200076                 i < BLK_SIZE && size > 0;
200077                 i++, j++, n++, size--, offset++)
200078              {
200079                block[i] = data_buffer[j];
200080              }
200081            block = blk_ata (device, DEV_WRITE, n_blk, block);
200082            if (block == NULL)
200083              {
200084                errset (errno);
200085                return (m);       // Up to the last
200086                // block written.
200087              }
200088            if (size)
200089              {
200090                n_blk = offset / BLK_SIZE;
200091                i = offset % BLK_SIZE;
200092                block = blk_ata (device, DEV_READ, n_blk,
200093                                 NULL);
200094                if (block == NULL)
200095                  {
200096                    errset (errno);
200097                    return (m);   // Up to the last
200098                    // block written.
200099                  }
200100              }
200101          }
200102      }
200103    //
200104    // Everything was right, so 'n' is valid for both
200105    // read or write.
200106    //
200107    return (n);
200108  }
```

94.3.2 kernel/dev/dev_dm.c

«

Si veda la sezione 93.4.2.

```
210001  #include <sys/os32.h>
210002  #include <kernel/dev.h>
210003  #include <kernel/driver/ata.h>
210004  #include <kernel/lib_k.h>
210005  #include <kernel/dm.h>
210006  #include <sys/types.h>
210007  #include <ctype.h>
210008  #include <errno.h>
210009  //-----------------------------------------------------------
210010  ssize_t
210011  dev_dm (pid_t pid, dev_t device, int rw, off_t offset,
210012          void *buffer, size_t size, int *eof)
210013  {
210014    int dev_minor = minor (device);
210015    int p = dev_minor & 0x000F;
210016    int d = ((dev_minor & 0x00F0) >> 4);
210017    //
210018    // If it is a partition, must verify if such
210019    // partition exists.
210020    //
210021    if (p)
210022      {
210023        //
210024        // It is a partition.
210025        //
210026        if (dm_table[d].part[p].type == PART_TYPE_NONE)
210027          {
210028            errset (ENODEV);
210029            return ((ssize_t) - 1);
210030          }
210031      }
210032    //
210033    // Shift the offset to the start of partition.
210034    //
210035    offset += (dm_table[d].part[p].start);
210036    //
210037    // Call the right hardware driver.
210038    //
210039    switch (dm_table[d].type)
```



```
210040      {
210041      case DM_TYPE_ATA:
210042        return (dev_ata
210043                (pid, device, rw, offset, buffer, size, eof));
210044        break;
210045      default:
210046        errset (ENODEV);
210047        return ((ssize_t) - 1);
210048      }
210049  }
```

94.3.3 kernel/dev/dev_io.c

«

Si veda la sezione 93.4.1.

```
220001  #include <sys/os32.h>
220002  #include <kernel/dev.h>
220003  #include <sys/types.h>
220004  #include <errno.h>
220005  #include <kernel/ibm_i386.h>
220006  #include <kernel/proc.h>
220007  #include <string.h>
220008  #include <signal.h>
220009  #include <kernel/lib_k.h>
220010  #include <ctype.h>
220011  #include <kernel/driver/tty.h>
220012  //-----------------------------------------------------------
220013  ssize_t
220014  dev_io (pid_t pid, dev_t device, int rw, off_t offset,
220015          void *buffer, size_t size, int *eof)
220016  {
220017    int dev_major = major (device);
220018    if (rw != DEV_READ && rw != DEV_WRITE)
220019      {
220020        errset (EIO);
220021        return (-1);
220022      }
220023    switch (dev_major)
220024      {
220025      case DEV_MEM_MAJOR:
220026        return (dev_mem
220027                (pid, device, rw, offset, buffer, size, eof));
220028      case DEV_TTY_MAJOR:
220029        return (dev_tty
220030                (pid, device, rw, offset, buffer, size, eof));
220031      case DEV_CONSOLE_MAJOR:
220032        return (dev_tty
220033                (pid, device, rw, offset, buffer, size, eof));
220034      case DEV_DM_MAJOR:
220035        return (dev_dm
220036                (pid, device, rw, offset, buffer, size, eof));
220037      case DEV_KMEM_MAJOR:
220038        return (dev_kmem
220039                (pid, device, rw, offset, buffer, size, eof));
220040      default:
220041        errset (ENODEV);
220042        return (-1);
220043      }
220044  }
```

94.3.4 kernel/dev/dev_kmem.c

«

Si veda la sezione 93.4.4.

```
230001  #include <sys/os32.h>
230002  #include <kernel/dev.h>
230003  #include <sys/types.h>
230004  #include <errno.h>
230005  #include <kernel/memory.h>
230006  #include <kernel/proc.h>
230007  #include <kernel/net/arp.h>
230008  #include <kernel/net/route.h>
230009  #include <kernel/net.h>
230010  #include <string.h>
230011  #include <signal.h>
230012  #include <ctype.h>
230013  //-----------------------------------------------------------
230014  ssize_t
230015  dev_kmem (pid_t pid, dev_t device, int rw,
230016            off_t offset, void *buffer, size_t size, int *eof)
230017  {
230018    inode_t *inode;
230019    sb_t *sb;
230020    file_t *file;
230021    void *start;
230022    char *m;
230023    //
```

```
230024    // Only read is allowed.
230025    //
230026    if (rw != DEV_READ)
230027      {
230028        errset (EIO);     // I/O error.
230029        return ((ssize_t) - 1);
230030      }
230031    //
230032    // Only positive offset is allowed.
230033    //
230034    if (offset < 0)
230035      {
230036        errset (EIO);     // I/O error.
230037        return ((ssize_t) - 1);
230038      }
230039    //
230040    // Read is selected (and is the only access
230041    // allowed).
230042    //
230043    switch (device)
230044      {
230045      case DEV_KMEM_PS:
230046        //
230047        // Verify if the selected slot can be read.
230048        //
230049        if (offset >= PROCESS_MAX)
230050          {
230051            errset (EIO); // I/O error.
230052            return ((ssize_t) - 1);
230053          }
230054        //
230055        // Correct the size to be read.
230056        //
230057        if (sizeof (proc_t) < size)
230058          {
230059            size = sizeof (proc_t);
230060          }
230061        //
230062        // Get the pointer to the selected slot.
230063        //
230064        start = proc_reference ((pid_t) offset);
230065        break;
230066      case DEV_KMEM_MMP:
230067        //
230068        // Correct the size to be read.
230069        //
230070        if (offset >= (MEM_MAX_BLOCKS / 8))
230071          {
230072            *eof = 1;
230073            errset (EIO); // I/O error.
230074            return ((ssize_t) - 1);
230075          }
230076        //
230077        // Reduce size if necessary.
230078        //
230079        if ((offset + size) > (MEM_MAX_BLOCKS / 8))
230080          {
230081            size = ((MEM_MAX_BLOCKS / 8) - offset);
230082          }
230083        //
230084        // Get the pointer to the map: offset is not
230085        // taken
230086        // into consideration.
230087        //
230088        m = (char *) mb_reference ();
230089        //
230090        start = &m[offset];
230091        //
230092        break;
230093      case DEV_KMEM_SB:
230094        //
230095        // Verify if the selected slot can be read.
230096        //
230097        if (offset >= SB_MAX_SLOTS)
230098          {
230099            errset (EIO); // I/O error.
230100            return ((ssize_t) - 1);
230101          }
230102        //
230103        // Get a reference to the super block table.
230104        //
230105        sb = sb_reference (0);
230106        //
230107        // Correct the size to be read.
230108        //
230109        if (sizeof (sb_t) < size)
230110          {
230111            size = sizeof (sb_t);
230112          }
230113        //
230114        // Get the pointer to the selected super block
230115        // slot.
230116        //
230117        start = &sb[offset];
230118        break;
230119      case DEV_KMEM_INODE:
230120        //
230121        // Verify if the selected slot can be read.
230122        //
230123        if (offset >= INODE_MAX_SLOTS)
230124          {
230125            errset (EIO); // I/O error.
230126            return ((ssize_t) - 1);
230127          }
230128        //
230129        // Get a reference to the inode table.
230130        //
230131        inode = inode_reference (0, 0);
230132        //
230133        // Correct the size to be read.
230134        //
230135        if (sizeof (inode_t) < size)
230136          {
230137            size = sizeof (inode_t);
230138          }
230139        //
230140        // Get the pointer to the selected inode slot.
230141        //
230142        start = &inode[offset];
230143        break;
230144      case DEV_KMEM_FILE:
230145        //
230146        // Verify if the selected slot can be read.
230147        //
230148        if (offset >= FILE_MAX_SLOTS)
230149          {
230150            errset (EIO); // I/O error.
230151            return ((ssize_t) - 1);
230152          }
230153        //
230154        // Get a reference to the file table.
230155        //
230156        file = file_reference (0);
230157        //
230158        // Correct the size to be read.
230159        //
230160        if (sizeof (file_t) < size)
230161          {
230162            size = sizeof (file_t);
230163          }
230164        //
230165        // Get the pointer to the selected inode slot.
230166        //
230167        start = &file[offset];
230168        break;
230169      case DEV_KMEM_ARP:
230170        //
230171        // Verify if the selected slot can be read.
230172        //
230173        if (offset >= ARP_MAX_ITEMS)
230174          {
230175            errset (EIO); // I/O error.
230176            return ((ssize_t) - 1);
230177          }
230178        //
230179        // Correct the size to be read.
230180        //
230181        if (sizeof (arp_t) < size)
230182          {
230183            size = sizeof (arp_t);
230184          }
230185        //
230186        // Get the pointer to the selected ARP item.
230187        //
230188        start = &arp_table[offset];
230189        break;
230190      case DEV_KMEM_NET:
230191        //
230192        // Verify if the selected slot can be read.
230193        //
230194        if (offset >= NET_MAX_DEVICES)
230195          {
230196            errset (EIO); // I/O error.
230197            return ((ssize_t) - 1);
```

```
230198          }
230199        //
230200        // Correct the size to be read.
230201        //
230202        if (sizeof (net_t) < size)
230203          {
230204            size = sizeof (net_t);
230205          }
230206        //
230207        // Get the pointer to the selected NET table
230208        // item.
230209        //
230210        start = &net_table[offset];
230211        break;
230212      case DEV_KMEM_ROUTE:
230213        //
230214        // Verify if the selected slot can be read.
230215        //
230216        if (offset >= ROUTE_MAX_ROUTES)
230217          {
230218            errset (EIO); // I/O error.
230219            return ((ssize_t) - 1);
230220          }
230221        //
230222        // Correct the size to be read.
230223        //
230224        if (sizeof (route_t) < size)
230225          {
230226            size = sizeof (route_t);
230227          }
230228        //
230229        // Get the pointer to the selected NET table
230230        // item.
230231        //
230232        start = &route_table[offset];
230233        break;
230234      default:
230235        errset (ENODEV);  // No such device.
230236        return ((ssize_t) - 1);
230237      }
230238    //
230239    // At this point, data is ready to be copied to the
230240    // buffer.
230241    //
230242    memcpy (buffer, start, size);
230243    //
230244    // Return size read.
230245    //
230246    return (size);
230247  }
```

94.3.5 kernel/dev/dev_mem.c

«

Si veda la sezione 93.4.5.

```
240001  #include <sys/os32.h>
240002  #include <kernel/dev.h>
240003  #include <sys/types.h>
240004  #include <errno.h>
240005  #include <kernel/memory.h>
240006  #include <kernel/ibm_i386.h>
240007  #include <kernel/proc.h>
240008  #include <string.h>
240009  #include <signal.h>
240010  #include <kernel/lib_k.h>
240011  #include <ctype.h>
240012  //-------------------------------------------------------------
240013  ssize_t
240014  dev_mem (pid_t pid, dev_t device, int rw, off_t offset,
240015           void *buffer, size_t size, int *eof)
240016  {
240017    uint8_t *buffer08 = (uint8_t *) buffer;
240018    uint16_t *buffer16 = (uint16_t *) buffer;
240019    ssize_t n;
240020
240021    if (device == DEV_MEM)        // DEV_MEM
240022      {
240023        if (rw == DEV_READ)
240024          {
240025            memcpy (buffer, (void *) (int) offset, size);
240026            n = size;
240027          }
240028        else
240029          {
240030            if (pid == 0)
240031              {
240032                memcpy ((void *) (int) offset, buffer, size);
```



```
240033                n = size;
240034              }
240035            else
240036              {
240037                k_printf
240038                  ("kernel alert: only the kernel "
240039                   "can write the memory where it "
240040                   "likes!\n");
240041                errset (EIO);     // I/O error.
240042                return ((ssize_t) - 1);
240043              }
240044          }
240045      }
240046    else if (device == DEV_NULL)  // DEV_NULL
240047      {
240048        n = 0;
240049      }
240050    else if (device == DEV_ZERO)  // DEV_ZERO
240051      {
240052        if (rw == DEV_READ)
240053          {
240054            for (n = 0; n < size; n++)
240055              {
240056                buffer08[n] = 0;
240057              }
240058          }
240059        else
240060          {
240061            n = 0;
240062          }
240063      }
240064    else if (device == DEV_PORT)  // DEV_PORT
240065      {
240066        if (rw == DEV_READ)
240067          {
240068            if (size == 1)
240069              {
240070                buffer08[0] = in_8 (offset);
240071                n = 1;
240072              }
240073            else if (size == 2)
240074              {
240075                buffer16[0] = in_16 (offset);
240076                n = 2;
240077              }
240078            else
240079              {
240080                n = 0;
240081              }
240082          }
240083        else
240084          {
240085            if (size == 1)
240086              {
240087                out_8 (offset, buffer08[0]);
240088              }
240089            else if (size == 2)
240090              {
240091                out_16 (offset, buffer16[0]);
240092                n = 2;
240093              }
240094            else
240095              {
240096                n = 0;
240097              }
240098          }
240099      }
240100    else
240101      {
240102        errset (ENODEV);
240103        return ((ssize_t) - 1);
240104      }
240105    return (n);
240106  }
```

94.3.6 kernel/dev/dev_tty.c

«

Si veda la sezione 93.4.6.

```
250001  #include <sys/os32.h>
250002  #include <kernel/dev.h>
250003  #include <sys/types.h>
250004  #include <errno.h>
250005  #include <kernel/memory.h>
250006  #include <kernel/ibm_i386.h>
250007  #include <kernel/proc.h>
250008  #include <string.h>
```

```
250009 #include <signal.h>
250010 #include <kernel/lib_k.h>
250011 #include <ctype.h>
250012 #include <kernel/driver/tty.h>
250013 //-------------------------------------------------------
250014 ssize_t
250015 dev_tty (pid_t pid, dev_t device, int rw, off_t offset,
250016          void *buffer, size_t size, int *eof)
250017 {
250018   uint8_t *buffer08 = (uint8_t *) buffer;
250019   ssize_t n;
250020   proc_t *ps;
250021   int key;
250022   //
250023   // Get process. Variable 'ps' will be 'NULL' if the
250024   // process ID is
250025   // not valid.
250026   //
250027   ps = proc_reference (pid);
250028   //
250029   // Convert 'DEV_TTY' with the controlling terminal
250030   // for the process.
250031   //
250032   if (device == DEV_TTY)
250033     {
250034       device = ps->device_tty;
250035       //
250036       // As a last resort, use the generic
250037       // 'DEV_CONSOLE'.
250038       //
250039       if (device == DEV_UNDEFINED || device == DEV_TTY)
250040         {
250041           device = DEV_CONSOLE;
250042         }
250043     }
250044   //
250045   // Convert 'DEV_CONSOLE' to the currently active
250046   // console.
250047   //
250048   if (device == DEV_CONSOLE)
250049     {
250050       device = tty_console ((dev_t) 0);
250051       //
250052       // As a last resort, use the first console:
250053       // 'DEV_CONSOLE0'.
250054       //
250055       if (device == DEV_UNDEFINED || device == DEV_TTY)
250056         {
250057           device = DEV_CONSOLE0;
250058         }
250059     }
250060   //
250061   // Read or write.
250062   //
250063   if (rw == DEV_READ)
250064     {
250065       for (n = 0; n < size; n++)
250066         {
250067           key = tty_read (device);
250068           if (key == 0 && n == 0)
250069             {
250070               //
250071               // A single line contains zero: this is
250072               // made by a VEOF
250073               // character (^d), that is, the input is
250074               // closed,
250075               // so return zero read and EOF.
250076               //
250077               *eof = 1;
250078               return (0);
250079             }
250080           else if (key == -1 && n == 0)
250081             {
250082               //
250083               // At the moment, there is just nothing
250084               // to read.
250085               //
250086               errset (EAGAIN);
250087               return (-1);
250088             }
250089           else if (key == -1 && n > 0)
250090             {
250091               //
250092               // Finished to read.
250093               //
250094               break;
250095             }
```



```
250096           else
250097             {
250098               buffer08[n] = key;
250099             }
250100         }
250101     }
250102   else
250103     {
250104       for (n = 0; n < size; n++)
250105         {
250106           tty_write (device, (int) buffer08[n]);
250107         }
250108     }
250109   return (n);
250110 }
```

94.4 os32: «kernel/dm.h»

Si veda la sezione 93.5.

```
260001 #ifndef _KERNEL_DM_H
260002 #define _KERNEL_DM_H    1
260003 //-------------------------------------------------------
260004 #include <stdint.h>
260005 #include <sys/types.h>
260006 #include <kernel/part.h>
260007 //-------------------------------------------------------
260008 #define DM_MAX_DEVICES   4
260009 #define DM_TYPE_NONE     0
260010 #define DM_TYPE_ATA      1
260011 //
260012 typedef struct
260013 {
260014   int type;
260015   void *table;
260016   struct
260017   {
260018     off_t start;
260019     size_t size;
260020     uint8_t type;
260021   } part[PART_MAX + 1];
260022 } dm_t;
260023 //
260024 extern dm_t dm_table[DM_MAX_DEVICES];
260025 //-------------------------------------------------------
260026 void dm_init (void);
260027 //-------------------------------------------------------
260028 #endif
```

94.4.1 kernel/dm/dm_init.c

Si veda la sezione 93.5.

```
270001 #include <kernel/dm.h>
270002 #include <kernel/part.h>
270003 #include <kernel/driver/ata.h>
270004 #include <kernel/lib_k.h>
270005 #include <stdint.h>
270006 #include <errno.h>
270007 //-------------------------------------------------------------
270008 void
270009 dm_init (void)
270010 {
270011   int d;
270012   int a;
270013   int p;
270014   ata_sector_t sector_buffer;
270015   part_t *part;
270016   int status;
270017   //
270018   // Reset the data-memory table.
270019   //
270020   for (d = 0; d < DM_MAX_DEVICES; d++)
270021     {
270022       dm_table[d].type = DM_TYPE_NONE;
270023       dm_table[d].table = NULL;
270024       dm_table[d].part[0].start = 0;
270025       dm_table[d].part[0].size = 0;
270026       dm_table[d].part[0].type = PART_TYPE_NO_PART;
270027       for (p = 0; p < PART_MAX; p++)
270028         {
270029           dm_table[d].part[p + 1].start = 0;
270030           dm_table[d].part[p + 1].size = 0;
270031           dm_table[d].part[p + 1].type = PART_TYPE_NONE;
270032         }
270033     }
270034   //
270035   // Reset data-memory index.
270036   //
270037   d = 0;
270038   //
270039   // Init ATA devices.
270040   //
270041   ata_init ();
270042   //
270043   // Assign ATA devices to the first data-memory
270044   // items.
270045   //
270046   for (a = 0; a < ATA_MAX_DEVICES; a++)
270047     {
270048       if (ata_table[a].present == 0)
270049         {
270050           //
270051           // Current data-memory device will be
270052           // used for the next ATA device, if any.
270053           //
270054           continue;
270055         }
270056       //
270057       // Show something.
270058       //
270059       k_printf ("[%s] ATA drive=%i total sectors=%i\n",
270060                 __func__, a, (int) ata_table[d].sectors);
270061       //
270062       dm_table[d].type = DM_TYPE_ATA;
270063       dm_table[d].table = &ata_table[a];
270064       dm_table[d].part[0].start = 0;
270065       dm_table[d].part[0].size = ata_table[a].sectors;
270066       dm_table[d].part[0].type = PART_TYPE_NO_PART;
270067       //
270068       // Read partitions.
270069       //
270070       status = ata_read_sector (a, 0, &sector_buffer);
270071       //
270072       if (status)
270073         {
270074           errset (errno);
270075           k_perror (NULL);
270076         }
270077       else
270078         {
270079           part =
270080             (((void *) &sector_buffer) + PART_TABLE_OFF);
270081           //
270082           for (p = 0; p < PART_MAX; p++)
270083             {
270084               //
270085               dm_table[d].part[p + 1].start =
270086                 part->l_start * ATA_SECTOR_SIZE;
270087               dm_table[d].part[p + 1].size =
270088                 part->size * ATA_SECTOR_SIZE;
270089               dm_table[d].part[p + 1].type = part->type;
270090               //
270091               // Show info.
270092               //
270093               if (part->type != 0)
270094                 {
270095                   k_printf ("[%s] partition type=%02x "
270096                             "start sector=%i "
270097                             "total sectors=%i\n",
270098                             __func__, (int) part->type,
270099                             (int) part->l_start,
270100                             (int) part->size);
270101                 }
270102               //
270103               part++;
270104             }
270105         }
270106       //
270107       // Next data-memory device.
270108       //
270109       d++;
270110     }
270111 }
```

94.4.2 kernel/dm/dm_public.c

Si veda la sezione 93.5.

```
280001 #include <kernel/dm.h>
280002 //-------------------------------------------------------------
280003 dm_t dm_table[DM_MAX_DEVICES];
```

94.4.3 kernel/driver/ata.h

Si veda la sezione 93.2.

```
290001 #ifndef _KERNEL_DRIVER_ATA_H
290002 #define _KERNEL_DRIVER_ATA_H    1
```

```
290003 //-------------------------------------------------------
290004 #include <stdint.h>
290005 #include <sys/types.h>
290006 #include <time.h>
290007 //-------------------------------------------------------
290008 //
290009 // I/O ports, used to access ATA bus registers. These
290010 // I/O ports are different for every ATA bus.
290011 //
290012 #define ATA0_DATA          0x1F0        // r/w
290013 #define ATA0_FEATURE       0x1F1        // -/w
290014 #define ATA0_ERROR         0x1F1        // r/-
290015 #define ATA0_COUNT         0x1F2        // r/w
290016 #define ATA0_LOW           0x1F3        // r/w
290017 #define ATA0_MID           0x1F4        // r/w
290018 #define ATA0_HIGH          0x1F5        // r/w
290019 #define ATA0_DEVICE        0x1F6        // r/w
290020 #define ATA0_COMMAND       0x1F7        // -/w
290021 #define ATA0_STATUS        0x1F7        // r/- regular
290022                                         // status
290023 #define ATA0_CONTROL       0x3F6        // -/w
290024 #define ATA0_ALTERNATE     0x3F6        // w/-
290025                                         // alternate
290026                                         // status
290027 //
290028 #define ATA1_DATA          0x170        // r/w
290029 #define ATA1_FEATURE       0x171        // -/w
290030 #define ATA1_ERROR         0x171        // r/-
290031 #define ATA1_COUNT         0x172        // r/w
290032 #define ATA1_LOW           0x173        // r/w
290033 #define ATA1_MID           0x174        // r/w
290034 #define ATA1_HIGH          0x175        // r/w
290035 #define ATA1_DEVICE        0x176        // r/w
290036 #define ATA1_COMMAND       0x177        // -/w
290037 #define ATA1_STATUS        0x177        // r/- regular
290038                                         // status
290039 #define ATA1_CONTROL       0x376        // -/w
290040 #define ATA1_ALTERNATE     0x376        // w/-
290041                                         // alternate
290042                                         // status
290043 //
290044 #define ATA2_DATA          0x1E8        // r/w
290045 #define ATA2_FEATURE       0x1E9        // -/w
290046 #define ATA2_ERROR         0x1E9        // r/-
290047 #define ATA2_COUNT         0x1EA        // r/w
290048 #define ATA2_LOW           0x1EB        // r/w
290049 #define ATA2_MID           0x1EC        // r/w
290050 #define ATA2_HIGH          0x1ED        // r/w
290051 #define ATA2_DEVICE        0x1EE        // r/w
290052 #define ATA2_COMMAND       0x1EF        // -/w
290053 #define ATA2_STATUS        0x1EF        // r/- regular
290054                                         // status
290055 #define ATA2_CONTROL       0x3E6        // -/w
290056 #define ATA2_ALTERNATE     0x3E6        // w/-
290057                                         // alternate
290058                                         // status
290059 //
290060 #define ATA3_DATA          0x168        // r/w
290061 #define ATA3_FEATURE       0x169        // -/w
290062 #define ATA3_ERROR         0x169        // r/-
290063 #define ATA3_COUNT         0x16A        // r/w
290064 #define ATA3_LOW           0x16B        // r/w
290065 #define ATA3_MID           0x16C        // r/w
290066 #define ATA3_HIGH          0x16D        // r/w
290067 #define ATA3_DEVICE        0x16E        // r/w
290068 #define ATA3_COMMAND       0x16F        // -/w
290069 #define ATA3_STATUS        0x16F        // r/- regular
290070                                         // status
290071 #define ATA3_CONTROL       0x366        // -/w
290072 #define ATA3_ALTERNATE     0x366        // w/-
290073                                         // alternate
290074                                         // status
290075 //
290076 // Status register flags (regular or alternate).
290077 //
290078 #define ATA_STATUS_BSY            0x80  // Busy
290079 #define ATA_STATUS_DRDY           0x40  // Ready
290080 #define ATA_STATUS_DF             0x20  // Drive Fault
290081 #define ATA_STATUS_DRQ            0x08  // Data
290082                                         // request
290083 #define ATA_STATUS_ERR            0x01  // Error
290084 //
290085 // Values to put to the device register
290086 //
290087 #define ATA_DEVICE_CHS          0x00
290088 #define ATA_DEVICE_LBA          0x40
290089 #define ATA_DEVICE_MASTER       0x00
```

```
290090 #define ATA_DEVICE_SLAVE        0x10
290091 //
290092 // Values to put to the command register
290093 //
290094 #define ATA_COMMAND_IDENTIFY_DEVICE      0xEC
290095 #define ATA_COMMAND_READ_SECTORS         0x20
290096 #define ATA_COMMAND_WRITE_SECTORS        0x30
290097 #define ATA_COMMAND_FLUSH_CACHE          0xE7
290098 //
290099 // Values to put to the control register
290100 // (device control register).
290101 //
290102 #define ATA_CONTROL_HOB         0x80
290103 #define ATA_CONTROL_SRST        0x04     // Software
290104                                          // reset.
290105 #define ATA_CONTROL_NIEN        0x01     // No
290106                                          // Interrupt
290107                                          // enabled.
290108 //
290109 //
290110 //
290111 #define ATA_MAX_DEVICES         8        // Fixed.
290112 #define ATA_SECTOR_SIZE         512      // Fixed.
290113 #define ATA_TIMEOUT  \
290114     ((clock_t) (CLOCKS_PER_SEC * 1))    // 1 s
290115 #define ATA_TIMEOUT_FLUSH \
290116     ((clock_t) (CLOCKS_PER_SEC * 10))   // 10 s
290117 //
290118 //
290119 //
290120 typedef struct
290121 {
290122   unsigned short r_data;
290123   unsigned short r_feature;
290124   unsigned short r_error;
290125   unsigned short r_count;
290126   unsigned short r_low;
290127   unsigned short r_mid;
290128   unsigned short r_high;
290129   unsigned short r_device;
290130   unsigned short r_command;
290131   unsigned short r_status;
290132   unsigned short r_control;
290133   unsigned short r_alternate;
290134   unsigned char bus;
290135   unsigned char target;
290136   unsigned char present;
290137   uint16_t id[ATA_SECTOR_SIZE / 2];
290138   unsigned int sectors;
290139 } ata_t;
290140 //
290141 typedef struct
290142 {
290143   char byte[ATA_SECTOR_SIZE];
290144 } ata_sector_t;
290145 //
290146 extern ata_t ata_table[ATA_MAX_DEVICES];
290147 //-------------------------------------------------------
290148 void ata_init (void);
290149 void ata_reset (int drive);
290150 int ata_valid (int drive);
290151 //-------------------------------------------------------
290152 int ata_cmd_identify_device (int drive, void *buffer);
290153 int ata_cmd_read_sectors (int drive,
290154                           unsigned int sector,
290155                           unsigned char count,
290156                           void *buffer);
290157 int ata_cmd_write_sectors (int drive,
290158                            unsigned int sector,
290159                            unsigned char count,
290160                            void *buffer);
290161 int ata_device (int drive, unsigned int sector);
290162 //-------------------------------------------------------
290163 int ata_rdy (int drive, clock_t timeout);
290164 int ata_drq (int drive, clock_t timeout);
290165 int ata_lba28 (int drive, unsigned int sector,
290166                unsigned char count);
290167 //-------------------------------------------------------
290168 #define ata_read_sector(drive, sector, buffer)  \
290169     (ata_cmd_read_sectors ((int) drive, \
290170      (unsigned int) sector, \
290171      (unsigned char) 1, (void *) buffer))
290172 //
290173 #define ata_write_sector(drive, sector, buffer)  \
290174     (ata_cmd_write_sectors ((int) drive, \
290175      (unsigned int) sector, \
290176      (unsigned char) 1, (void *) buffer))
```

```
290177 //-------------------------------------------------------
290178 #endif
```

94.4.4 kernel/driver/ata/ata_cmd_identify_device.c

Si veda la sezione 93.2.

```
300001 #include <kernel/driver/ata.h>
300002 #include <kernel/lib_k.h>
300003 #include <kernel/ibm_i386.h>
300004 #include <stdint.h>
300005 #include <errno.h>
300006 //-------------------------------------------------------
300007 int
300008 ata_cmd_identify_device (int drive, void *buffer)
300009 {
300010   unsigned char status;
300011   int s;
300012   int i;
300013   uint16_t *id = buffer;
300014   //
300015   // Register 'device'.
300016   //
300017   s = ata_device (drive, 0);
300018   if (s < 0)
300019     {
300020       errset (errno);
300021       return (-1);
300022     }
300023   //
300024   // Send 'command'
300025   //
300026   out_8 (ata_table[drive].r_command,
300027          ATA_COMMAND_IDENTIFY_DEVICE);
300028   //
300029   // Read the regular status port.
300030   //
300031   status = in_8 (ata_table[drive].r_status);
300032   //
300033   // If the status is zero, there is no drive.
300034   //
300035   if (status == 0)
300036     {
300037       //
300038       // Clear the 'id[]' array and return.
300039       //
300040       for (i = 0; i < (ATA_SECTOR_SIZE / 2); i++)
300041         {
300042           id[i] = 0;
300043         }
300044       return (0);
300045     }
300046   //
300047   // Wait for the drive ready to send data.
300048   //
300049   s = ata_drq (drive, ATA_TIMEOUT);
300050   if (s < 0)
300051     {
300052       errset (errno);
300053       return (-1);
300054     }
300055   //
300056   // Read data.
300057   //
300058   for (i = 0; i < (ATA_SECTOR_SIZE / 2); i++)
300059     {
300060       id[i] = in_16 (ata_table[drive].r_data);
300061     }
300062   //
300063   // Return.
300064   //
300065   return (0);
300066 }
```

94.4.5 kernel/driver/ata/ata_cmd_read_sectors.c

Si veda la sezione 93.2.

```
310001 #include <kernel/driver/ata.h>
310002 #include <kernel/lib_k.h>
310003 #include <kernel/ibm_i386.h>
310004 #include <stdint.h>
310005 #include <errno.h>
310006 //-------------------------------------------------------
310007 int
310008 ata_cmd_read_sectors (int drive, unsigned int sector,
310009                       unsigned char count, void *buffer)
310010 {
310011   int s;
310012   int i;
310013   int c;
310014   uint16_t *destination = buffer;
310015   //
310016   // Set LBA 28 parameters.
310017   //
310018   s = ata_lba28 (drive, sector, count);
310019   if (s < 0)
310020     {
310021       errset (errno);
310022       return (-1);
310023     }
310024   //
310025   // Send 'command'
310026   //
310027   out_8 (ata_table[drive].r_command,
310028          ATA_COMMAND_READ_SECTORS);
310029   //
310030   // Parameter 'count' equal to zero means 256
310031   // sectors.
310032   //
310033   if (count == 0)
310034     {
310035       c = 256;
310036     }
310037   else
310038     {
310039       c = count;
310040     }
310041   //
310042   // Read 'c' sectors.
310043   //
310044   for (; c > 0; c--)
310045     {
310046       s = ata_drq (drive, ATA_TIMEOUT);
310047       if (s < 0)
310048         {
310049           errset (errno);
310050           return (-1);
310051         }
310052       //
310053       // Read sector.
310054       //
310055       for (i = 0; i < (ATA_SECTOR_SIZE / 2); i++)
310056         {
310057           destination[i] = in_16 (ata_table[drive].r_data);
310058         }
310059     }
310060   //
310061   // Wait that the device returns ready.
310062   //
310063   s = ata_rdy (drive, ATA_TIMEOUT);
310064   if (s < 0)
310065     {
310066       errset (errno);
310067       return (-1);
310068     }
310069   //
310070   // Return.
310071   //
310072   return (0);
310073 }
```

94.4.6 kernel/driver/ata/ata_cmd_write_sectors.c

Si veda la sezione 93.2.

```
320001 #include <kernel/driver/ata.h>
320002 #include <kernel/lib_k.h>
320003 #include <kernel/ibm_i386.h>
320004 #include <stdint.h>
320005 #include <errno.h>
320006 //-------------------------------------------------------
320007 int
320008 ata_cmd_write_sectors (int drive, unsigned int sector,
320009                        unsigned char count, void *buffer)
320010 {
320011   int s;
320012   int i;
320013   int c;
320014   uint16_t *source = buffer;
320015   //
320016   // Set LBA 28 parameters.
320017   //
320018   s = ata_lba28 (drive, sector, count);
```

```
320019    if (s < 0)
320020      {
320021        errset (errno);
320022        return (-1);
320023      }
320024    //
320025    // Send 'command'
320026    //
320027    out_8 (ata_table[drive].r_command,
320028           ATA_COMMAND_WRITE_SECTORS);
320029    //
320030    // Parameter 'count' equal to zero means 256
320031    // sectors.
320032    //
320033    if (count == 0)
320034      {
320035        c = 256;
320036      }
320037    else
320038      {
320039        c = count;
320040      }
320041    //
320042    // Read 'c' sectors.
320043    //
320044    for (; c > 0; c--)
320045      {
320046        s = ata_drq (drive, ATA_TIMEOUT);
320047        if (s < 0)
320048          {
320049            errset (errno);
320050            return (-1);
320051          }
320052        //
320053        // Write sector.
320054        //
320055        for (i = 0; i < (ATA_SECTOR_SIZE / 2); i++)
320056          {
320057            out_16 (ata_table[drive].r_data, source[i]);
320058          }
320059      }
320060    //
320061    // Wait that the device returns ready.
320062    //
320063    s = ata_rdy (drive, ATA_TIMEOUT);
320064    if (s < 0)
320065      {
320066        errset (errno);
320067        return (-1);
320068      }
320069    //
320070    // Now flush cache.
320071    //
320072    out_8 (ata_table[drive].r_command,
320073           ATA_COMMAND_FLUSH_CACHE);
320074    //
320075    // Then wait that the device returns ready.
320076    //
320077    s = ata_rdy (drive, ATA_TIMEOUT_FLUSH);
320078    if (s < 0)
320079      {
320080        errset (errno);
320081        return (-1);
320082      }
320083    //
320084    // Return.
320085    //
320086    return (0);
320087  }
```

94.4.7 kernel/driver/ata/ata_device.c

«

Si veda la sezione 93.2.

```
330001  #include <kernel/driver/ata.h>
330002  #include <kernel/lib_s.h>
330003  #include <kernel/ibm_i386.h>
330004  #include <stdint.h>
330005  #include <errno.h>
330006  //-----------------------------------------------------------
330007  int
330008  ata_device (int drive, unsigned int sector)
330009  {
330010    unsigned char device;
330011    int s;
330012    //
330013    // Verify 'drive' argument.
```



```
330014    //
330015    s = ata_valid (drive);
330016    if (s < 0)
330017      {
330018        errset (EINVAL);
330019        return (-1);
330020      }
330021    //
330022    // Start building the 'device' register.
330023    //
330024    device = 0;
330025    //
330026    // The access will be LBA: no CHS at all here!
330027    //
330028    device |= ATA_DEVICE_LBA;
330029    //
330030    // Set the device number, relative to the bus.
330031    //
330032    device |= ata_table[drive].target;
330033    //
330034    // Put the highest four bits of the sector number,
330035    // that can use at most 28 bits.
330036    //
330037    device |= ((sector & 0x0F000000) >> 24);
330038    //
330039    // Must select the new drive.
330040    //
330041    out_8 (ata_table[drive].r_device, device);
330042    //
330043    // Wait for selected drive ready.
330044    //
330045    s = ata_rdy (drive, ATA_TIMEOUT);
330046    if (s < 0)
330047      {
330048        errset (errno);
330049        return (-1);
330050      }
330051    //
330052    // Ok.
330053    //
330054    return (0);
330055  }
```

94.4.8 kernel/driver/ata/ata_drq.c

«

Si veda la sezione 93.2.

```
340001  #include <kernel/driver/ata.h>
340002  #include <kernel/lib_s.h>
340003  #include <kernel/ibm_i386.h>
340004  #include <stdint.h>
340005  #include <errno.h>
340006  //-----------------------------------------------------------
340007  int
340008  ata_drq (int drive, clock_t timeout)
340009  {
340010    clock_t time_start;
340011    clock_t time_now;
340012    clock_t time_elapsed;
340013    int status;
340014    //
340015    // The timeout value must be at least two.
340016    //
340017    if (timeout < 2)
340018      {
340019        timeout = 2;
340020      }
340021    //
340022    // Get the status register.
340023    //
340024    time_elapsed = 0;
340025    time_start = s_clock ((pid_t) 0);
340026    while (time_elapsed < timeout)
340027      {
340028        time_now = s_clock ((pid_t) 0);
340029        time_elapsed = time_now - time_start;
340030        //
340031        status = in_8 (ata_table[drive].r_status);
340032        //
340033        // Is it BSY?
340034        //
340035        if (status & ATA_STATUS_BSY)
340036          {
340037            //
340038            // Read the status again.
340039            //
340040            continue;
```

```
340041          }
340042        //
340043        // No more busy, but check for errors.
340044        //
340045        if (status & ATA_STATUS_DF)
340046          {
340047            k_printf ("[%s] ERROR: drive %i fault\n",
340048                      __func__, drive);
340049            ata_reset (drive);
340050            errset (E_HARDWARE_FAULT);
340051            return (-1);
340052          }
340053        //
340054        if (status & ATA_STATUS_ERR)
340055          {
340056            k_printf ("[%s] ERROR: drive %i error\n",
340057                      __func__, drive);
340058            ata_reset (drive);
340059            errset (E_DRIVER_FAULT);
340060            return (-1);
340061          }
340062        //
340063        // Now check for the DRQ.
340064        //
340065        if (status & ATA_STATUS_DRQ)
340066          {
340067            //
340068            // Ok.
340069            //
340070            return (0);
340071          }
340072      }
340073    //
340074    // Sorry: time out!
340075    //
340076    k_printf ("[%s] ERROR: drive %i timeout\n", __func__,
340077              drive);
340078    errset (ETIME);
340079    return (-1);
340080  }
```

94.4.9 kernel/driver/ata/ata_init.c

«

Si veda la sezione 93.2.

```
350001  #include <kernel/driver/ata.h>
350002  #include <kernel/lib_k.h>
350003  #include <kernel/ibm_i386.h>
350004  #include <stdint.h>
350005  #include <errno.h>
350006  //-----------------------------------------------------------
350007  void
350008  ata_init (void)
350009  {
350010    unsigned char status;
350011    int s;
350012    int d;
350013    //
350014    // Set bus numbers and I/O ports for each possible
350015    // drive.
350016    // I/O ports are related to the bus, so every couple
350017    // of
350018    // drive has the same ports.
350019    //
350020    if (ATA_MAX_DEVICES > 0)
350021      {
350022        ata_table[0].bus = 0;
350023        ata_table[0].r_data = ATA0_DATA;
350024        ata_table[0].r_feature = ATA0_FEATURE;
350025        ata_table[0].r_error = ATA0_ERROR;
350026        ata_table[0].r_count = ATA0_COUNT;
350027        ata_table[0].r_low = ATA0_LOW;
350028        ata_table[0].r_mid = ATA0_MID;
350029        ata_table[0].r_high = ATA0_HIGH;
350030        ata_table[0].r_device = ATA0_DEVICE;
350031        ata_table[0].r_command = ATA0_COMMAND;
350032        ata_table[0].r_status = ATA0_STATUS;
350033        ata_table[0].r_control = ATA0_CONTROL;
350034        ata_table[0].r_alternate = ATA0_ALTERNATE;
350035        ata_table[0].target = ATA_DEVICE_MASTER;
350036        //
350037        ata_table[1].bus = 0;
350038        ata_table[1].r_data = ATA0_DATA;
350039        ata_table[1].r_feature = ATA0_FEATURE;
350040        ata_table[1].r_error = ATA0_ERROR;
350041        ata_table[1].r_count = ATA0_COUNT;
350042        ata_table[1].r_low = ATA0_LOW;
```



```
350043        ata_table[1].r_mid = ATA0_MID;
350044        ata_table[1].r_high = ATA0_HIGH;
350045        ata_table[1].r_device = ATA0_DEVICE;
350046        ata_table[1].r_command = ATA0_COMMAND;
350047        ata_table[1].r_status = ATA0_STATUS;
350048        ata_table[1].r_control = ATA0_CONTROL;
350049        ata_table[1].r_alternate = ATA0_ALTERNATE;
350050        ata_table[1].target = ATA_DEVICE_SLAVE;
350051      }
350052    //
350053    if (ATA_MAX_DEVICES > 2)
350054      {
350055        ata_table[2].bus = 1;
350056        ata_table[2].r_data = ATA1_DATA;
350057        ata_table[2].r_feature = ATA1_FEATURE;
350058        ata_table[2].r_error = ATA1_ERROR;
350059        ata_table[2].r_count = ATA1_COUNT;
350060        ata_table[2].r_low = ATA1_LOW;
350061        ata_table[2].r_mid = ATA1_MID;
350062        ata_table[2].r_high = ATA1_HIGH;
350063        ata_table[2].r_device = ATA1_DEVICE;
350064        ata_table[2].r_command = ATA1_COMMAND;
350065        ata_table[2].r_status = ATA1_STATUS;
350066        ata_table[2].r_control = ATA1_CONTROL;
350067        ata_table[2].r_alternate = ATA1_ALTERNATE;
350068        ata_table[2].target = ATA_DEVICE_MASTER;
350069        //
350070        ata_table[3].bus = 1;
350071        ata_table[3].r_data = ATA1_DATA;
350072        ata_table[3].r_feature = ATA1_FEATURE;
350073        ata_table[3].r_error = ATA1_ERROR;
350074        ata_table[3].r_count = ATA1_COUNT;
350075        ata_table[3].r_low = ATA1_LOW;
350076        ata_table[3].r_mid = ATA1_MID;
350077        ata_table[3].r_high = ATA1_HIGH;
350078        ata_table[3].r_device = ATA1_DEVICE;
350079        ata_table[3].r_command = ATA1_COMMAND;
350080        ata_table[3].r_status = ATA1_STATUS;
350081        ata_table[3].r_control = ATA1_CONTROL;
350082        ata_table[3].r_alternate = ATA1_ALTERNATE;
350083        ata_table[3].target = ATA_DEVICE_SLAVE;
350084      }
350085    //
350086    if (ATA_MAX_DEVICES > 4)
350087      {
350088        ata_table[4].bus = 2;
350089        ata_table[4].r_data = ATA2_DATA;
350090        ata_table[4].r_feature = ATA2_FEATURE;
350091        ata_table[4].r_error = ATA2_ERROR;
350092        ata_table[4].r_count = ATA2_COUNT;
350093        ata_table[4].r_low = ATA2_LOW;
350094        ata_table[4].r_mid = ATA2_MID;
350095        ata_table[4].r_high = ATA2_HIGH;
350096        ata_table[4].r_device = ATA2_DEVICE;
350097        ata_table[4].r_command = ATA2_COMMAND;
350098        ata_table[4].r_status = ATA2_STATUS;
350099        ata_table[4].r_control = ATA2_CONTROL;
350100        ata_table[4].r_alternate = ATA2_ALTERNATE;
350101        ata_table[4].target = ATA_DEVICE_MASTER;
350102        //
350103        ata_table[5].bus = 2;
350104        ata_table[5].r_data = ATA2_DATA;
350105        ata_table[5].r_feature = ATA2_FEATURE;
350106        ata_table[5].r_error = ATA2_ERROR;
350107        ata_table[5].r_count = ATA2_COUNT;
350108        ata_table[5].r_low = ATA2_LOW;
350109        ata_table[5].r_mid = ATA2_MID;
350110        ata_table[5].r_high = ATA2_HIGH;
350111        ata_table[5].r_device = ATA2_DEVICE;
350112        ata_table[5].r_command = ATA2_COMMAND;
350113        ata_table[5].r_status = ATA2_STATUS;
350114        ata_table[5].r_control = ATA2_CONTROL;
350115        ata_table[5].r_alternate = ATA2_ALTERNATE;
350116        ata_table[5].target = ATA_DEVICE_SLAVE;
350117      }
350118    //
350119    if (ATA_MAX_DEVICES > 6)
350120      {
350121        ata_table[6].bus = 3;
350122        ata_table[6].r_data = ATA3_DATA;
350123        ata_table[6].r_feature = ATA3_FEATURE;
350124        ata_table[6].r_error = ATA3_ERROR;
350125        ata_table[6].r_count = ATA3_COUNT;
350126        ata_table[6].r_low = ATA3_LOW;
350127        ata_table[6].r_mid = ATA3_MID;
350128        ata_table[6].r_high = ATA3_HIGH;
350129        ata_table[6].r_device = ATA3_DEVICE;
```

```
350130        ata_table[6].r_command = ATA3_COMMAND;
350131        ata_table[6].r_status = ATA3_STATUS;
350132        ata_table[6].r_control = ATA3_CONTROL;
350133        ata_table[6].r_alternate = ATA3_ALTERNATE;
350134        ata_table[6].target = ATA_DEVICE_MASTER;
350135        //
350136        ata_table[7].bus = 3;
350137        ata_table[7].r_data = ATA3_DATA;
350138        ata_table[7].r_feature = ATA3_FEATURE;
350139        ata_table[7].r_error = ATA3_ERROR;
350140        ata_table[7].r_count = ATA3_COUNT;
350141        ata_table[7].r_low = ATA3_LOW;
350142        ata_table[7].r_mid = ATA3_MID;
350143        ata_table[7].r_high = ATA3_HIGH;
350144        ata_table[7].r_device = ATA3_DEVICE;
350145        ata_table[7].r_command = ATA3_COMMAND;
350146        ata_table[7].r_status = ATA3_STATUS;
350147        ata_table[7].r_control = ATA3_CONTROL;
350148        ata_table[7].r_alternate = ATA3_ALTERNATE;
350149        ata_table[7].target = ATA_DEVICE_SLAVE;
350150      }
350151    //
350152    // Scan and eliminate buses with no drive at all.
350153    //
350154    for (d = 0; d < ATA_MAX_DEVICES; d += 2)
350155      {
350156        status = in_8 (ata_table[d].r_status);
350157        if (status == 0xFF)
350158          {
350159            ata_table[d].present = 0;
350160            ata_table[d + 1].present = 0;
350161          }
350162        else
350163          {
350164            ata_table[d].present = 1;
350165            ata_table[d + 1].present = 1;
350166          }
350167      }
350168    //
350169    // Identify drives.
350170    //
350171    for (d = 0; d < ATA_MAX_DEVICES; d++)
350172      {
350173        if (ata_table[d].present == 0)
350174          {
350175            continue;
350176          }
350177        //
350178        // Register 'device'.
350179        //
350180        s = ata_device (d, 0);
350181        if (s < 0)
350182          {
350183            errset (errno);
350184            k_perror (NULL);
350185            ata_table[d].present = 0;
350186            continue;
350187          }
350188        //
350189        // Send command 'IDENTIFY DEVICE'
350190        //
350191        s =
350192          ata_cmd_identify_device (d, &(ata_table[d].id[0]));
350193        if (s < 0)
350194          {
350195            ata_table[d].present = 0;
350196            continue;
350197          }
350198        //
350199        // Verify again if the drive is present: the
350200        // function might have found that it does not
350201        // exist.
350202        //
350203        if (ata_table[d].present == 0)
350204          {
350205            continue;
350206          }
350207        //
350208        // Find total sectors (for 28 bit LBA access).
350209        // It is written
350210        // inside the integer formed by 'identity[60]'
350211        // and
350212        // 'identity[61]', considering it in little
350213        // endian mode.
350214        // It is taken the pointer to 'identity[60]',
350215        // transformed
350216        // into a pointer to a 32 bit integer, and then
```



```
350217        // dereferenced
350218        // again.
350219        //
350220        ata_table[d].sectors
350221          = *((uint32_t *) & (ata_table[d].id[60]));
350222        //
350223        // Check if the size value is right.
350224        //
350225        if (ata_table[d].sectors == 0)
350226          {
350227            ata_table[d].present = 0;
350228          }
350229        else
350230          {
350231            //
350232            // Show info.
350233            //
350234            k_printf ("[%s] ATA drive %i size %i Kib\n",
350235                      __func__, d, ata_table[d].sectors / 2);
350236          }
350237      }
350238  }
```

94.4.10 kernel/driver/ata/ata_lba28.c

Si veda la sezione 93.2.

```
360001  #include <kernel/driver/ata.h>
360002  #include <kernel/lib_s.h>
360003  #include <kernel/ibm_i386.h>
360004  #include <stdint.h>
360005  #include <errno.h>
360006  //-----------------------------------------------------------
360007  int
360008  ata_lba28 (int drive, unsigned int sector,
360009             unsigned char count)
360010  {
360011    int s;
360012    unsigned char low;
360013    unsigned char mid;
360014    unsigned char high;
360015    //
360016    // Register 'device'.
360017    //
360018    s = ata_device (drive, sector);
360019    if (s < 0)
360020      {
360021        k_perror (NULL);
360022
360023        errset (errno);
360024        return (-1);
360025      }
360026    //
360027    // Register 'control', to set nIEN.
360028    //
360029    out_8 (ata_table[drive].r_control, ATA_CONTROL_NIEN);
360030    //
360031    // Register 'feature'. not used.
360032    //
360033    out_8 (ata_table[drive].r_feature, 0);
360034    //
360035    // Register 'count'
360036    //
360037    out_8 (ata_table[drive].r_count, count);
360038    //
360039    // Registers 'low', 'mid', 'high'.
360040    //
360041    low = (sector & 0x000000FF);
360042    mid = ((sector & 0x0000FF00) >> 8);
360043    high = ((sector & 0x00FF0000) >> 16);
360044    //
360045    out_8 (ata_table[drive].r_low, low);
360046    out_8 (ata_table[drive].r_mid, mid);
360047    out_8 (ata_table[drive].r_high, high);
360048    //
360049    // Ok.
360050    //
360051    return (0);
360052  }
```

94.4.11 kernel/driver/ata/ata_public.c

Si veda la sezione 93.2.

```
370001  #include <kernel/driver/ata.h>
370002  //-----------------------------------------------------------
370003  ata_t ata_table[ATA_MAX_DEVICES];
```

```
370004 //-------------------------------------------------------
```

94.4.12 kernel/driver/ata/ata_rdy.c

Si veda la sezione 93.2.

```
380001 #include <kernel/driver/ata.h>
380002 #include <kernel/lib_s.h>
380003 #include <kernel/ibm_i386.h>
380004 #include <stdint.h>
380005 #include <errno.h>
380006 //-------------------------------------------------------
380007 int
380008 ata_rdy (int drive, clock_t timeout)
380009 {
380010   clock_t time_start;
380011   clock_t time_now;
380012   clock_t time_elapsed;
380013   unsigned char status;
380014   //
380015   // The timeout value must be at least two.
380016   //
380017   if (timeout < 2)
380018     {
380019       timeout = 2;
380020     }
380021   //
380022   // Get the status register.
380023   //
380024   time_elapsed = 0;
380025   time_start = s_clock ((pid_t) 0);
380026   while (time_elapsed < timeout)
380027     {
380028       time_now = s_clock ((pid_t) 0);
380029       time_elapsed = time_now - time_start;
380030       //
380031       status = in_8 (ata_table[drive].r_status);
380032       //
380033       // Is it BSY?
380034       //
380035       if (status & ATA_STATUS_BSY)
380036         {
380037           //
380038           // Read the status again.
380039           //
380040           continue;
380041         }
380042       //
380043       // No more busy, but check for errors.
380044       //
380045       if (status & ATA_STATUS_DF)
380046         {
380047           k_printf ("[%s] ERROR: drive %i fault\n",
380048                     __func__, drive);
380049           ata_reset (drive);
380050           errset (E_HARDWARE_FAULT);
380051           return (-1);
380052         }
380053       //
380054       if (status & ATA_STATUS_ERR)
380055         {
380056           k_printf ("[%s] ERROR: drive %i error\n",
380057                     __func__, drive);
380058           ata_reset (drive);
380059           errset (E_DRIVER_FAULT);
380060           return (-1);
380061         }
380062       //
380063       // Otherwise: ok.
380064       //
380065       return (0);
380066     }
380067   //
380068   // Sorry: time out!
380069   //
380070   k_printf ("[%s] ERROR: drive %i timeout\n", __func__,
380071             drive);
380072   errset (ETIME);
380073   return (-1);
380074 }
```

94.4.13 kernel/driver/ata/ata_reset.c

Si veda la sezione 93.2.

```
390001 #include <kernel/driver/ata.h>
390002 #include <kernel/lib_s.h>
```



```
390003 #include <kernel/ibm_i386.h>
390004 #include <stdint.h>
390005 #include <errno.h>
390006 //-------------------------------------------------------
390007 void
390008 ata_reset (int drive)
390009 {
390010   out_8 (ata_table[drive].r_control, ATA_CONTROL_SRST);
390011   out_8 (ata_table[drive].r_control, 0);
390012 }
```

94.4.14 kernel/driver/ata/ata_valid.c

Si veda la sezione 93.2.

```
400001 #include <kernel/driver/ata.h>
400002 #include <errno.h>
400003 //-------------------------------------------------------
400004 int
400005 ata_valid (int drive)
400006 {
400007   //
400008   // Verify if 'drive' argument is possible.
400009   //
400010   if (drive < 0 || drive >= ATA_MAX_DEVICES)
400011     {
400012       errset (EINVAL);
400013       return (-1);
400014     }
400015   //
400016   // Verify if 'drive' is present at the moment.
400017   //
400018   if (ata_table[drive].present == 0)
400019     {
400020       errset (E_NO_MEDIUM);
400021       return (-1);
400022     }
400023   //
400024   // OK.
400025   //
400026   return (0);
400027 }
```

94.4.15 kernel/driver/kbd.h

Si veda la sezione 93.10.

```
410001 #ifndef _KERNEL_DRIVER_KBD_H
410002 #define _KERNEL_DRIVER_KBD_H     1
410003 //-------------------------------------------------------
410004 #include <stdbool.h>
410005 //-------------------------------------------------------
410006 //
410007 // Keyboard status.
410008 //
410009 typedef struct
410010 {
410011   bool shift;
410012   bool shift_lock;
410013   bool ctrl;
410014   bool alt;
410015   bool echo;
410016   unsigned char key;
410017   unsigned char map1[128];
410018   unsigned char map2[128];
410019 } kbd_t;
410020 //
410021 extern kbd_t kbd;
410022 //-------------------------------------------------------
410023 void kbd_load (void);
410024 void kbd_isr (void);
410025 //-------------------------------------------------------
410026 #endif
```

94.4.16 kernel/driver/kbd/kbd_isr.c

Si veda la sezione 93.10.

```
420001 #include <kernel/driver/kbd.h>
420002 #include <kernel/ibm_i386.h>
420003 //-------------------------------------------------------
420004 void
420005 kbd_isr (void)
420006 {
420007   //
420008   // Get a character from the keyboard channel.
420009   //
```

```
420010    unsigned char scancode = (int) in_8 (0x60);
420011    //
420012    // Check for [Shift], [Shift-Lock], [Ctrl] and [Alt]
420013    // keys.
420014    //
420015    switch (scancode)
420016      {
420017      case 0x2A:
420018        kbd.shift = 1;
420019        break;
420020      case 0x36:
420021        kbd.shift = 1;
420022        break;
420023      case 0xAA:
420024        kbd.shift = 0;
420025        break;
420026      case 0xB6:
420027        kbd.shift = 0;
420028        break;
420029      case 0x1D:
420030        kbd.ctrl = 1;
420031        break;
420032      case 0x9D:
420033        kbd.ctrl = 0;
420034        break;
420035      case 0x38:
420036        kbd.alt = 1;
420037        break;
420038      case 0xB8:
420039        kbd.alt = 0;
420040        break;
420041      case 0x3A:
420042        kbd.shift_lock = !kbd.shift_lock;
420043        break;
420044      }
420045    //
420046    // Check for usual keys, but only if 'kbd.key' is
420047    // currently empty.
420048    //
420049    if (scancode <= 127 && kbd.ctrl && kbd.key == 0)
420050      {
420051        //
420052        // Something was pressed, combined with [Ctrl].
420053        //
420054        switch (kbd.map1[scancode])
420055          {
420056          case 'a':
420057            kbd.key = 0x01;
420058            break;        // SOH
420059          case 'b':
420060            kbd.key = 0x02;
420061            break;        // STX
420062          case 'c':
420063            kbd.key = 0x03;
420064            break;        // ETX
420065          case 'd':
420066            kbd.key = 0x04;
420067            break;        // EOT
420068          case 'e':
420069            kbd.key = 0x05;
420070            break;        // ENQ
420071          case 'f':
420072            kbd.key = 0x06;
420073            break;        // ACK
420074          case 'g':
420075            kbd.key = 0x07;
420076            break;        // BEL
420077          case 'h':
420078            kbd.key = 0x08;
420079            break;        // BS
420080          case 'i':
420081            kbd.key = 0x09;
420082            break;        // HT
420083          case 'j':
420084            kbd.key = 0x0A;
420085            break;        // LF
420086          case 'k':
420087            kbd.key = 0x0B;
420088            break;        // VT
420089          case 'l':
420090            kbd.key = 0x0C;
420091            break;        // FF
420092          case 'm':
420093            kbd.key = 0x0D;
420094            break;        // CR
420095          case 'n':
420096            kbd.key = 0x0E;
```



```
420097            break;        // SO
420098          case 'o':
420099            kbd.key = 0x0F;
420100            break;        // SI
420101          case 'p':
420102            kbd.key = 0x10;
420103            break;        // DLE
420104          case 'q':
420105            kbd.key = 0x11;
420106            break;        // DC1
420107          case 'r':
420108            kbd.key = 0x12;
420109            break;        // DC2
420110          case 's':
420111            kbd.key = 0x13;
420112            break;        // DC3
420113          case 't':
420114            kbd.key = 0x14;
420115            break;        // DC4
420116          case 'u':
420117            kbd.key = 0x15;
420118            break;        // NAK
420119          case 'v':
420120            kbd.key = 0x16;
420121            break;        // SYN
420122          case 'w':
420123            kbd.key = 0x17;
420124            break;        // ETB
420125          case 'x':
420126            kbd.key = 0x18;
420127            break;        // CAN
420128          case 'y':
420129            kbd.key = 0x19;
420130            break;        // EM
420131          case 'z':
420132            kbd.key = 0x1A;
420133            break;        // SUB
420134          case '[':
420135            kbd.key = 0x1B;
420136            break;        // ESC
420137          case '\\':
420138            kbd.key = 0x1C;
420139            break;        // FS
420140          case ']':
420141            kbd.key = 0x1D;
420142            break;        // GS
420143          case '^':
420144            kbd.key = 0x1E;
420145            break;        // RS
420146          case '_':
420147            kbd.key = 0x1F;
420148            break;        // US
420149          }
420150      }
420151    else if (scancode <= 127 && kbd.key == 0
420152             && kbd.map1[scancode] != 0)
420153      {
420154        //
420155        // Something was pressed, but no [Ctrl] is
420156        // there.
420157        //
420158        if (kbd.shift || kbd.shift_lock)
420159          {
420160            kbd.key = kbd.map2[scancode];
420161          }
420162        else
420163          {
420164            kbd.key = kbd.map1[scancode];
420165          }
420166      }
420167  }
```

94.4.17 kernel/driver/kbd/kbd_load.c

Si veda la sezione 93.10.

```
430001  #include <kernel/driver/kbd.h>
430002  //-----------------------------------------------------------
430003  void
430004  kbd_load (void)
430005  {
430006    int i;
430007    //
430008    // Reset the map.
430009    //
430010    for (i = 0; i <= 127; i++)
430011      {
```

```
430012        kbd.map1[i] = 0;
430013        kbd.map2[i] = 0;
430014      }
430015    //
430016    // Association for an italian map, but only with
430017    // ASCII characters
430018    // (there are no accented letters).
430019    //
430020    kbd.map1[1] = 27;
430021    kbd.map2[1] = 27;     // ESC
430022    kbd.map1[2] = '1';
430023    kbd.map2[2] = '!';
430024    kbd.map1[3] = '2';
430025    kbd.map2[3] = '"';
430026    kbd.map1[4] = '3';
430027    kbd.map2[4] = 'L';    // 3, £
430028    kbd.map1[5] = '4';
430029    kbd.map2[5] = '$';
430030    kbd.map1[6] = '5';
430031    kbd.map2[6] = '%';
430032    kbd.map1[7] = '6';
430033    kbd.map2[7] = '&';
430034    kbd.map1[8] = '7';
430035    kbd.map2[8] = '/';
430036    kbd.map1[9] = '8';
430037    kbd.map2[9] = '(';
430038    kbd.map1[10] = '9';
430039    kbd.map2[10] = ')';
430040    kbd.map1[11] = '0';
430041    kbd.map2[11] = '=';
430042    kbd.map1[12] = '\'';
430043    kbd.map2[12] = '?';
430044    kbd.map1[13] = 'i';
430045    kbd.map2[13] = '^';   // ì, ^
430046    kbd.map1[14] = '\b';
430047    kbd.map2[14] = '\b';  // Backspace
430048    kbd.map1[15] = '\t';
430049    kbd.map2[15] = '\t';
430050    kbd.map1[16] = 'q';
430051    kbd.map2[16] = 'Q';
430052    kbd.map1[17] = 'w';
430053    kbd.map2[17] = 'W';
430054    kbd.map1[18] = 'e';
430055    kbd.map2[18] = 'E';
430056    kbd.map1[19] = 'r';
430057    kbd.map2[19] = 'R';
430058    kbd.map1[20] = 't';
430059    kbd.map2[20] = 'T';
430060    kbd.map1[21] = 'y';
430061    kbd.map2[21] = 'Y';
430062    kbd.map1[22] = 'u';
430063    kbd.map2[22] = 'U';
430064    kbd.map1[23] = 'i';
430065    kbd.map2[23] = 'I';
430066    kbd.map1[24] = 'o';
430067    kbd.map2[24] = 'O';
430068    kbd.map1[25] = 'p';
430069    kbd.map2[25] = 'P';
430070    kbd.map1[26] = '[';
430071    kbd.map2[26] = '{';   // è, é
430072    kbd.map1[27] = ']';
430073    kbd.map2[27] = '}';   // +, *
430074    kbd.map1[28] = '\n';
430075    kbd.map2[28] = '\n';  // Enter
430076    kbd.map1[30] = 'a';
430077    kbd.map2[30] = 'A';
430078    kbd.map1[31] = 's';
430079    kbd.map2[31] = 'S';
430080    kbd.map1[32] = 'd';
430081    kbd.map2[32] = 'D';
430082    kbd.map1[33] = 'f';
430083    kbd.map2[33] = 'F';
430084    kbd.map1[34] = 'g';
430085    kbd.map2[34] = 'G';
430086    kbd.map1[35] = 'h';
430087    kbd.map2[35] = 'H';
430088    kbd.map1[36] = 'j';
430089    kbd.map2[36] = 'J';
430090    kbd.map1[37] = 'k';
430091    kbd.map2[37] = 'K';
430092    kbd.map1[38] = 'l';
430093    kbd.map2[38] = 'L';
430094    kbd.map1[39] = '@';
430095    kbd.map2[39] = '@';   // ò, ç
430096    kbd.map1[40] = '#';
430097    kbd.map2[40] = '#';   // à, °
430098    kbd.map1[41] = '\\';
```



```
430099    kbd.map2[41] = '|';
430100    kbd.map1[43] = 'u';
430101    kbd.map2[43] = 'U';   // ù, §
430102    kbd.map1[44] = 'z';
430103    kbd.map2[44] = 'Z';
430104    kbd.map1[45] = 'x';
430105    kbd.map2[45] = 'X';
430106    kbd.map1[46] = 'c';
430107    kbd.map2[46] = 'C';
430108    kbd.map1[47] = 'v';
430109    kbd.map2[47] = 'V';
430110    kbd.map1[48] = 'b';
430111    kbd.map2[48] = 'B';
430112    kbd.map1[49] = 'n';
430113    kbd.map2[49] = 'N';
430114    kbd.map1[50] = 'm';
430115    kbd.map2[50] = 'M';
430116    kbd.map1[51] = ',';
430117    kbd.map2[51] = ';';
430118    kbd.map1[52] = '.';
430119    kbd.map2[52] = ':';
430120    kbd.map1[53] = '-';
430121    kbd.map2[53] = '_';
430122    kbd.map1[56] = '<';
430123    kbd.map2[56] = '>';
430124    kbd.map1[57] = ' ';
430125    kbd.map2[57] = ' ';
430126    //
430127    kbd.map1[55] = '*';
430128    kbd.map2[55] = '*';
430129    kbd.map1[71] = '7';
430130    kbd.map2[71] = '7';
430131    kbd.map1[72] = '8';
430132    kbd.map2[72] = '8';
430133    kbd.map1[73] = '9';
430134    kbd.map2[73] = '9';
430135    kbd.map1[74] = '-';
430136    kbd.map2[74] = '-';
430137    kbd.map1[75] = '4';
430138    kbd.map2[75] = '4';
430139    kbd.map1[76] = '5';
430140    kbd.map2[76] = '5';
430141    kbd.map1[77] = '6';
430142    kbd.map2[77] = '6';
430143    kbd.map1[78] = '+';
430144    kbd.map2[78] = '+';
430145    kbd.map1[79] = '1';
430146    kbd.map2[79] = '1';
430147    kbd.map1[80] = '2';
430148    kbd.map2[80] = '2';
430149    kbd.map1[81] = '3';
430150    kbd.map2[81] = '3';
430151    kbd.map1[82] = '0';
430152    kbd.map2[82] = '0';
430153    kbd.map1[83] = '.';
430154    kbd.map2[83] = '.';
430155    kbd.map1[92] = '/';
430156    kbd.map2[92] = '/';
430157    kbd.map1[96] = '\n';
430158    kbd.map2[96] = '\n';  // Enter
430159  }
```

94.4.18 kernel/driver/kbd/kbd_public.c

Si veda la sezione 93.10.

```
440001  #include <kernel/driver/kbd.h>
440002  //-----------------------------------------------------------
440003  kbd_t kbd;
```

94.4.19 kernel/driver/nic/ne2k.h

Si veda la sezione 93.16.

```
450001  #ifndef _KERNEL_DRIVER_NIC_NE2K_H
450002  #define _KERNEL_DRIVER_NIC_NE2K_H    1
450003  //-----------------------------------------------------------
450004  #include <stdint.h>
450005  #include <sys/types.h>
450006  #include <unistd.h>
450007  #include <time.h>
450008  //-----------------------------------------------------------
450009  //
450010  // Command register, available on all pages.
450011  //
450012  #define NE2K_CR         0x00    // rw
450013  //
```

```
450014 // Page 0 registers
450015 //
450016 #define NE2K_CLDA0      0x01    // r-
450017 #define NE2K_PSTART     0x01    // -w
450018 #define NE2K_CLDA1      0x02    // r-
450019 #define NE2K_PSTOP      0x02    // -w
450020 #define NE2K_BNRY       0x03    // rw
450021 #define NE2K_TSR        0x04    // r-
450022 #define NE2K_TPSR       0x04    // -w
450023 #define NE2K_NCR        0x05    // r-
450024 #define NE2K_TBCR0      0x05    // -w
450025 #define NE2K_FIFO       0x06    // r-
450026 #define NE2K_TBCR1      0x06    // -w
450027 #define NE2K_ISR        0x07    // rw
450028 #define NE2K_CRDA0      0x08    // r-
450029 #define NE2K_RSAR0      0x08    // -w
450030 #define NE2K_CRDA1      0x09    // r-
450031 #define NE2K_RSAR1      0x09    // -w
450032 #define NE2K_RBCR0      0x0A    // -w
450033 #define NE2K_RBCR1      0x0B    // -w
450034 #define NE2K_RSR        0x0C    // r-
450035 #define NE2K_RCR        0x0C    // -w
450036 #define NE2K_CNTR0      0x0D    // r-
450037 #define NE2K_TCR        0x0D    // -w
450038 #define NE2K_CNTR1      0x0E    // r-
450039 #define NE2K_DCR        0x0E    // -w
450040 #define NE2K_CNTR2      0x0F    // r-
450041 #define NE2K_IMR        0x0F    // -w
450042 //
450043 // Page 1 registers: all read and write
450044 //
450045 #define NE2K_PAR0       0x01    // rw
450046 #define NE2K_PAR1       0x02    // rw
450047 #define NE2K_PAR2       0x03    // rw
450048 #define NE2K_PAR3       0x04    // rw
450049 #define NE2K_PAR4       0x05    // rw
450050 #define NE2K_PAR5       0x06    // rw
450051 #define NE2K_CURR       0x07    // rw
450052 #define NE2K_MAR0       0x08    // rw
450053 #define NE2K_MAR1       0x09    // rw
450054 #define NE2K_MAR2       0x0A    // rw
450055 #define NE2K_MAR3       0x0B    // rw
450056 #define NE2K_MAR4       0x0C    // rw
450057 #define NE2K_MAR5       0x0D    // rw
450058 #define NE2K_MAR6       0x0E    // rw
450059 #define NE2K_MAR7       0x0F    // rw
450060 //
450061 // Page 2 registers: read version of the same registers
450062 // inside page 1:
450063 //
450064 //      NE2K_PSTART     0x01 // -w
450065 //      NE2K_PSTOP      0x02 // -w
450066 //      --              0x03
450067 //      NE2K_TPSR       0x04 // -w
450068 //      --              0x05
450069 //      --              0x06
450070 //      --              0x07
450071 //      --              0x08
450072 //      --              0x09
450073 //      --              0x0A
450074 //      --              0x0B
450075 //      NE2K_RCR        0x0C // -w
450076 //      NE2K_TCR        0x0D // -w
450077 //      NE2K_DCR        0x0E // -w
450078 //      NE2K_IMR        0x0F // -w
450079 //
450080 // Extra registers, outside pages.
450081 //
450082 #define NE2K_DATA       0x10    // Data port.
450083 #define NE2K_RESET      0x1f    // Reset port.
450084 //
450085 // DMA buffer structure.
450086 //
450087 // 0x0000 .-------------------------.
450088 //        |                         |
450089 //        |                         |
450090 //        |           ???           |
450091 //        |                         |
450092 //        |                         |
450093 // 0x4000 |-------------------------|
450094 //        | transmit buffer ring [1] |
450095 // 0x4600 |-------------------------|
450096 //        |                         |
450097 //        |   receive buffer ring   |
450098 //        |                         |
450099 // 0xC000 |-------------------------|
450100 //        |                         |
```



```
450101 //        |           ???           |
450102 //        |                         |
450103 //        |                         |
450104 //        '-------------------------' 0xFFFF
450105 //
450106 // [1] 0x600 is equal to 1536, that is the max space
450107 //     that a packet can occupy inside the ring buffer.
450108 //     So, there is the place for a
450109 //     single transmission packet.
450110 //
450111 // Local DMA addresses:
450112 //
450113 #define NE2K_TX_START  0x40     // Means: 0x4000
450114 #define NE2K_TX_STOP   0x46     // Means: 0x4600
450115
450116 #define NE2K_RX_START  0x46     // Means: 0x4600
450117 #define NE2K_RX_STOP   0xC0     // Means: 0xC000
450118 //
450119 // Transmit buffer (remote DMA).
450120 //
450121 #define NE2K_TX_BUFFER NE2K_TX_START
450122 //
450123 // SA-PROM (Station Address PROM) size
450124 //
450125 #define NE2K_SAPROM_SIZE 32     // 32 bytes
450126 //-------------------------------------------------------
450127 int ne2k_check (uintptr_t io);
450128 int ne2k_isr (uintptr_t io);
450129 int ne2k_isr_expect (uintptr_t io, unsigned int isr_expect);
450130 int ne2k_reset (uintptr_t io, void *address);
450131 int ne2k_rx (uintptr_t io);
450132 int ne2k_rx_reset (uintptr_t io);
450133 int ne2k_tx (uintptr_t io, void *buffer, size_t size);
450134 //
450135 void ne2k_test2 (int eth);
450136
450137
450138 //-------------------------------------------------------
450139 #endif
```

94.4.20 kernel/driver/nic/ne2k/ne2k_check.c

Si veda la sezione 93.16.

```
460001 #include <kernel/driver/nic/ne2k.h>
460002 #include <kernel/ibm_i386.h>
460003 #include <errno.h>
460004 //-------------------------------------------------------
460005 int
460006 ne2k_check (uintptr_t io)
460007 {
460008   int status;
460009   int reg_00;
460010   int reg_0d;
460011   //
460012   // Read and save the command register (CR, 0x00): if
460013   // it is really the
460014   // command register, should not be equal to 0xFF.
460015   //
460016   reg_00 = in_8 (io + NE2K_CR);
460017   if (reg_00 == 0xFF)
460018     {
460019       errset (E_DRIVER_FAULT);
460020       return (-1);
460021     }
460022   //
460023   // Command register (CR)
460024   // .-------------------------------.
460025   // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
460026   // |-------------------------------|
460027   // | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0x61
460028   // '-------------------------------'
460029   // \___/ \_______/ |
460030   // | | STOP
460031   // | Abort/complete
460032   // | remote DMA
460033   // Register
460034   // page 1
460035   //
460036   // Stop and select page 1.
460037   //
460038   out_8 ((io + NE2K_CR), 0x61);
460039   //
460040   // Read, save and overwrite the register MAR5 (0x0D
460041   // at
460042   // page 1).
460043   //
```

```
460044    reg_0d = in_8 (io + NE2K_MAR5);
460045    out_8 (io + NE2K_MAR5, 0xFF);
460046    //
460047    // Command register (CR)
460048    // .-------------------------------.
460049    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
460050    // |-------------------------------|
460051    // | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
460052    // '-------------------------------'
460053    // \___/ \_______/ |
460054    // | | STOP
460055    // | Abort/complete
460056    // | remote DMA
460057    // Register
460058    // page 0
460059    //
460060    // Stop and select page 0.
460061    //
460062    out_8 ((io + NE2K_CR), 0x21);
460063    //
460064    // Read the tally counter 0 (CNTR0) to clear it.
460065    //
460066    in_8 (io + NE2K_CNTR0);
460067    //
460068    // Now the tally counter 0 (CNTR0) should be zero.
460069    //
460070    status = in_8 (io + NE2K_CNTR0);
460071    if (status)
460072      {
460073        //
460074        // The value obtained is not zero, so it is not
460075        // a NE2000 nic and the page change had probably
460076        // no effect. So, restore the values found
460077        // inside
460078        // 0x00 and 0x0D, without trying to change page.
460079        //
460080        out_8 (io, reg_00);
460081        out_8 ((io + 0x0D), reg_0d);
460082        errset (E_DRIVER_FAULT);
460083        return (-1);
460084      }
460085    //
460086    // Everything is ok: it might be a NE2000 nic.
460087    //
460088    return (0);
460089  }
```

94.4.21 kernel/driver/nic/ne2k/ne2k_isr.c

Si veda la sezione 93.16.

```
470001  #include <kernel/driver/pci.h>
470002  #include <kernel/driver/nic/ne2k.h>
470003  #include <kernel/ibm_i386.h>
470004  #include <errno.h>
470005  #include <kernel/lib_k.h>
470006  #include <kernel/lib_s.h>
470007  //-------------------------------------------------------
470008  int
470009  ne2k_isr (uintptr_t io)
470010  {
470011    int isr;
470012    //
470013    // Get ISR (interrupt status register).
470014    //
470015    isr = in_8 (io + NE2K_ISR);
470016    //
470017    //
470018    //
470019    if (isr & 0x01)
470020      {
470021        //
470022        // Frame received.
470023        //
470024        out_8 (io + NE2K_ISR, 0x01);
470025        ne2k_rx (io);
470026      }
470027    if (isr & 0x04)
470028      {
470029        //
470030        // Frame received with errors.
470031        //
470032        out_8 (io + NE2K_ISR, 0x04);
470033      }
470034    if (isr & 0x02)
470035      {
470036        //
470037        // Frame sent correctly.
470038        //
470039        ;
470040      }
470041    if (isr & 0x08)
470042      {
470043        //
470044        // Frame sent with errors.
470045        //
470046        ;
470047      }
470048    if (isr & 0x10)
470049      {
470050        k_printf ("OVERWRITE\n");
470051        out_8 (io + NE2K_ISR, 0x10);
470052        //
470053        // I don't understand if it works: Bochs just
470054        // don't accept
470055        // frames if they can make an overflow.
470056        //
470057        ne2k_rx (io);
470058        ne2k_rx_reset (io);
470059      }
470060    if (isr & 0x20)
470061      {
470062        //
470063        // Counter overflow.
470064        //
470065        out_8 (io + NE2K_ISR, 0x20);
470066      }
470067    if (isr & 0x40)
470068      {
470069        //
470070        // Remote DMA complete.
470071        //
470072        ;
470073      }
470074    if (isr & 0x80)
470075      {
470076        //
470077        // Reset status.
470078        //
470079        ;
470080      }
470081    //
470082    // End.
470083    //
470084    return (0);
470085  }
```

94.4.22 kernel/driver/nic/ne2k/ne2k_isr_expect.c

Si veda la sezione 93.16.

```
480001  #include <kernel/driver/nic/ne2k.h>
480002  #include <kernel/ibm_i386.h>
480003  #include <kernel/lib_k.h>
480004  #include <errno.h>
480005  #include <unistd.h>
480006  #include <time.h>
480007  //-------------------------------------------------------
480008  #define DEBUG 0
480009  //-------------------------------------------------------
480010  int
480011  ne2k_isr_expect (uintptr_t io, unsigned int isr_expect)
480012  {
480013    int retry = 5;
480014    int status;
480015    //
480016    //
480017    //
480018    for (; retry > 0; retry--)
480019      {
480020        status = in_8 (io + NE2K_ISR);
480021        //
480022        if (status & isr_expect)
480023          {
480024            //
480025            // Reset the bit found true and exit loop.
480026            //
480027            out_8 ((io + NE2K_ISR), isr_expect);
480028            return (0);
480029          }
480030      }
480031    //
480032    // If ISR is zero, we assume that it is ok.
480033    //
```

```
480034    if (status == 0)
480035      {
480036        if (DEBUG)
480037          {
480038            k_printf ("[isr=0x%02x expect=0x%02x]",
480039                      status, isr_expect);
480040            return (0);
480041          }
480042        return (0);
480043      }
480044    else
480045      {
480046        if (DEBUG)
480047          {
480048            //
480049            // It is not zero, but we prefer to let it
480050            // go...
480051            //
480052            k_printf ("[isr=0x%02x expect=0x%02x]",
480053                      status, isr_expect);
480054            return (0);
480055          }
480056        else
480057          {
480058            errset (E_DRIVER_FAULT);
480059            return (-1);
480060          }
480061      }
480062  }
```

94.4.23 kernel/driver/nic/ne2k/ne2k_reset.c

«

Si veda la sezione 93.16.

```
490001  #include <kernel/driver/pci.h>
490002  #include <kernel/driver/nic/ne2k.h>
490003  #include <kernel/ibm_i386.h>
490004  #include <errno.h>
490005  #include <kernel/lib_k.h>
490006  #include <kernel/lib_s.h>
490007  //-------------------------------------------------------------
490008  int
490009  ne2k_reset (uintptr_t io, void *address)
490010  {
490011    int status;
490012    int i;
490013    uint8_t sa_prom[NE2K_SAPROM_SIZE];
490014    uint8_t par[6];
490015    //
490016    // ------------------------------------------------
490017    // RESET
490018    // ------------------------------------------------
490019    //
490020    status = in_8 (io + NE2K_RESET);
490021    out_8 ((io + NE2K_RESET), 0xFF);
490022    out_8 ((io + NE2K_RESET), status);
490023    //
490024    // Interrupt status register (ISR)
490025    // .-------------------------------.
490026    // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
490027    // |-------------------------------|
490028    // | 1 | ? | ? | ? | ? | ? | ? | ? | 0x80
490029    // `-------------------------------'
490030    // |
490031    // Reset status
490032    //
490033    // Verify to have reset the NIC.
490034    //
490035    status = ne2k_isr_expect (io, 0x80);
490036    if (status)
490037      {
490038        errset (errno);
490039        return (-1);
490040      }
490041    //
490042    // Reset all ISR register flags.
490043    //
490044    out_8 ((io + NE2K_ISR), 0xFF);
490045    //
490046    // ------------------------------------------------
490047    // GET ETHERNET ADDRESS FROM SA-PROM (Station
490048    // Address PROM)
490049    // ------------------------------------------------
490050    //
490051    // Command register (CR)
490052    // .-------------------------------.
490053    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
```



```
490054    // |-------------------------------|
490055    // | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
490056    // `-------------------------------'
490057    // \_____/ \_______/ |
490058    // |  |  STOP
490059    // |  |
490060    // |  Abort/complete
490061    // |  remote DMA
490062    // |
490063    // Register
490064    // page 0
490065    //
490066    out_8 ((io + NE2K_CR), 0x21);
490067    //
490068    // Interrupt status register (ISR)
490069    // .-------------------------------.
490070    // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
490071    // |-------------------------------|
490072    // | 1 | ? | ? | ? | ? | ? | ? | ? | 0x80
490073    // `-------------------------------'
490074    // |
490075    // Reset status
490076    //
490077    // Verify to have reset the NIC.
490078    //
490079    status = ne2k_isr_expect (io, 0x80);
490080    if (status)
490081      {
490082        errset (errno);
490083        return (-1);
490084      }
490085    //
490086    // Data configuration register (DCR)
490087    // .-------------------------------.
490088    // | - |FT1|FT0|ARM| LS|LAS|BOS|WTS|
490089    // |-------------------------------|
490090    // | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0x48
490091    // `-------------------------------'
490092    // \_____/ : | : : :
490093    // | : | : : Byte DMA transfer
490094    // | : | : :
490095    // | : | : Little endian byte order
490096    // | : | :
490097    // | : | Dual 16 bit DMA mode
490098    // | : |
490099    // | : Loopback OFF (normal operation)
490100    // | :
490101    // | Send Command non executed: all frames removed
490102    // from
490103    // | Buffer Ring under program control
490104    // |
490105    // FIFO threshold 8 bytes
490106    //
490107    out_8 ((io + NE2K_DCR), 0x48);
490108    //
490109    // Reset remote byte count registers.
490110    //
490111    out_8 ((io + NE2K_RBCR0), 0x00);
490112    out_8 ((io + NE2K_RBCR1), 0x00);
490113    //
490114    // Disable interrupts with an empty mask.
490115    //
490116    out_8 ((io + NE2K_IMR), 0x00);
490117    //
490118    // Reset all ISR register flags.
490119    //
490120    out_8 ((io + NE2K_ISR), 0xFF);
490121    //
490122    // Receive configuration register (RCR)
490123    // .-------------------------------.
490124    // | - | - |MON|PRO| AM| AB| AR|SEP|
490125    // |-------------------------------|
490126    // | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0x20
490127    // `-------------------------------'
490128    // | : : : : :
490129    // | : : : : Frames with receive errors are
490130    // | : : : : rejected
490131    // | : : : :
490132    // | : : : Frames with fewer than 64 bytes rejected
490133    // | : : :
490134    // | : : Frames with broadcast destination rejected
490135    // | : : accepted
490136    // | : :
490137    // | : Frames with multicast destination address
490138    // | : not checked
490139    // | :
490140    // | Physical address of node must match the station
```

```
490141    // | address
490142    // |
490143    // Monitor mode: frames checked but not buffered to
490144    // memory
490145    //
490146    // Monitor mode.
490147    //
490148    out_8 ((io + NE2K_RCR), 0x20);
490149    //
490150    // Transmit configuration register (TCR)
490151    // .-------------------------------.
490152    // | - | - | - |OFST|ATD|LB1|LB0|CRC|
490153    // |-------------------------------|
490154    // | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0x02
490155    // '-------------------------------'
490156    // \_____/ :
490157    // | CRC appended by the transmitter
490158    // |
490159    // Internal loopback (mode 1)
490160    //
490161    // [Loopback is not supported by Bochs]
490162    //
490163    // Transmit loopback.
490164    //
490165    out_8 ((io + NE2K_TCR), 0x02);
490166    //
490167    // Remote byte count registers to NE2K_SAPROM_SIZE:
490168    // the bytes to be
490169    // read from SA-PROM.
490170    //
490171    out_8 ((io + NE2K_RBCR0), NE2K_SAPROM_SIZE);
490172    out_8 ((io + NE2K_RBCR1), (NE2K_SAPROM_SIZE >> 8));
490173    //
490174    // Set the remote DMA address to zero.
490175    //
490176    out_8 ((io + NE2K_RSAR0), 0x00);      // Must be
490177    // zero.
490178    out_8 ((io + NE2K_RSAR1), 0x00);
490179    //
490180    // Command register (CR)
490181    // .-------------------------------.
490182    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
490183    // |-------------------------------|
490184    // | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
490185    // '-------------------------------'
490186    // \___/ \_______/ |
490187    // | | START
490188    // | Read
490189    // |
490190    // Register
490191    // page 0
490192    //
490193    out_8 ((io + NE2K_CR), 0x0A);
490194    //
490195    // Save the SA-PROM content.
490196    //
490197    for (i = 0; i < NE2K_SAPROM_SIZE; i++)
490198      {
490199        sa_prom[i] = in_8 (io + NE2K_DATA);
490200      }
490201    //
490202    // Set NIC physical address from SA-PROM data.
490203    //
490204    par[0] = sa_prom[0];
490205    par[1] = sa_prom[2];
490206    par[2] = sa_prom[4];
490207    par[3] = sa_prom[6];
490208    par[4] = sa_prom[8];
490209    par[5] = sa_prom[10];
490210    //
490211    // Copy to the 'address' pointer.
490212    //
490213    if (address != NULL)
490214      {
490215        memcpy (address, par, (size_t) 6);
490216      }
490217    //
490218    // ------------------------------------------------
490219    // INITIALIZATION SEQUENCE
490220    // ------------------------------------------------
490221    //
490222    // Command register (CR)
490223    // .-------------------------------.
490224    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
490225    // |-------------------------------|
490226    // | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
490227    // '-------------------------------'
```

```
490228    // \_____/ \_________/ |
490229    // | | STOP
490230    // | |
490231    // | Abort/complete
490232    // | remote DMA
490233    // |
490234    // Register
490235    // page 0
490236    //
490237    out_8 ((io + NE2K_CR), 0x21);
490238    //
490239    // There is no need to check the ISR value. At this
490240    // point,
490241    // ISR might report a reset status or a remote DMA
490242    // complete.
490243    // Go to the DCR register.
490244    //
490245    // Data configuration register (DCR)
490246    // .-------------------------------.
490247    // | - |FT1|FT0|ARM| LS|LAS|BOS|WTS|
490248    // |-------------------------------|
490249    // | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0x48
490250    // '-------------------------------'
490251    // \_____/ : | : : :
490252    // | : | : : Byte DMA transfer
490253    // | : | : :
490254    // | : | : Little endian byte order
490255    // | : | :
490256    // | : | Dual 16 bit DMA mode
490257    // | : |
490258    // | : Loopback OFF (normal operation)
490259    // | :
490260    // | Send Command non executed: all frames removed
490261    // from
490262    // | Buffer Ring under program control
490263    // |
490264    // FIFO threshold 8 bytes
490265    //
490266    out_8 ((io + NE2K_DCR), 0x48);
490267    //
490268    // Reset remote byte count registers.
490269    //
490270    out_8 ((io + NE2K_RBCR0), 0x00);
490271    out_8 ((io + NE2K_RBCR1), 0x00);
490272    //
490273    // Receive configuration register (RCR)
490274    // .-------------------------------.
490275    // | - | - |MON|PRO| AM| AB| AR|SEP|
490276    // |-------------------------------|
490277    // | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0x04
490278    // '-------------------------------'
490279    // : : : | : :
490280    // : : : | : Frames with receive errors are
490281    // : : : | : rejected
490282    // : : : | :
490283    // : : : | Frames with fewer than 64 bytes rejected
490284    // : : : |
490285    // : : : Frames with broadcast destination address
490286    // : : : accepted
490287    // : : :
490288    // : : Frames with multicast destination address
490289    // : : not checked
490290    // : :
490291    // : Physical address of node must match the station
490292    // : address
490293    // :
490294    // Frames buffered to memory
490295    //
490296    // Normal operation and broadcast accepted.
490297    //
490298    out_8 ((io + NE2K_RCR), 0x04);
490299    //
490300    // Transmit page start (local DMA).
490301    //
490302    out_8 ((io + NE2K_TPSR), NE2K_TX_START);
490303    //
490304    // Transmit configuration register (TCR)
490305    // .-------------------------------.
490306    // | - | - | - |OFST|ATD|LB1|LB0|CRC|
490307    // |-------------------------------|
490308    // | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0x02
490309    // '-------------------------------'
490310    // \_____/ :
490311    // | CRC appended by the transmitter
490312    // |
490313    // Internal loopback (mode 1)
490314    //
```

```
490315    // [Loopback is not supported by Bochs]
490316    //
490317    // Transmit loopback.
490318    //
490319    out_8 ((io + NE2K_TCR), 0x02);
490320    //
490321    // Set receive buffer page start (local DMA).
490322    //
490323    out_8 ((io + NE2K_PSTART), NE2K_RX_START);
490324    //
490325    // Set boundary: the frame not yet read. At the
490326    // moment, it is the same
490327    // as the receive buffer page start.
490328    //
490329    out_8 ((io + NE2K_BNRY), NE2K_RX_START);
490330    //
490331    // Set receive buffer page stop (local DMA).
490332    //
490333    out_8 ((io + NE2K_PSTOP), NE2K_RX_STOP);
490334    //
490335    // Command register (CR)
490336    // .-----------------------------.
490337    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
490338    // |-----------------------------|
490339    // | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0x61
490340    // '-----------------------------'
490341    // \_____/ \_________/ |
490342    // | | STOP
490343    // | |
490344    // | Abort/complete
490345    // | remote DMA
490346    // |
490347    // Register
490348    // page 1
490349    //
490350    out_8 ((io + NE2K_CR), 0x61);
490351    //
490352    // Save physical address and multicast address.
490353    //
490354    out_8 ((io + NE2K_PAR0), par[0]);
490355    out_8 ((io + NE2K_PAR1), par[1]);
490356    out_8 ((io + NE2K_PAR2), par[2]);
490357    out_8 ((io + NE2K_PAR3), par[3]);
490358    out_8 ((io + NE2K_PAR4), par[4]);
490359    out_8 ((io + NE2K_PAR5), par[5]);
490360    //
490361    out_8 ((io + NE2K_MAR0), 0);
490362    out_8 ((io + NE2K_MAR1), 0);
490363    out_8 ((io + NE2K_MAR2), 0);
490364    out_8 ((io + NE2K_MAR3), 0);
490365    out_8 ((io + NE2K_MAR4), 0);
490366    out_8 ((io + NE2K_MAR5), 0);
490367    out_8 ((io + NE2K_MAR6), 0);
490368    out_8 ((io + NE2K_MAR7), 0);
490369    //
490370    // Set current page: the first frame to be saved
490371    // inside the receive
490372    // buffer. At the moment, it is the same as the
490373    // buffer page start.
490374    //
490375    out_8 ((io + NE2K_CURR), NE2K_RX_START);
490376    //
490377    // Command register (CR)
490378    // .-----------------------------.
490379    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
490380    // |-----------------------------|
490381    // | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0x22
490382    // '-----------------------------'
490383    // \_____/ \_________/ |
490384    // | | START
490385    // | |
490386    // | Abort/complete
490387    // | remote DMA
490388    // |
490389    // Register
490390    // page 0
490391    //
490392    out_8 ((io + NE2K_CR), 0x22);
490393    //
490394    // Reset all ISR register flags.
490395    //
490396    out_8 ((io + NE2K_ISR), 0xFF);
490397    //
490398    // ISR will be polled, but received packets will
490399    // fire the IRQ,
490400    // although it is not necessary. So the IMR
490401    // (Interrupt mask register)
```



```
490402    // is now set properly with the value 0x01.
490403    //
490404    // Interrupt mask register (IMR)
490405    // .-------------------------------------.
490406    // | -- |RDCE|CNTE|OVWE|TXEE|RXEE|PTXE|PRXE|
490407    // |-------------------------------------|
490408    // | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0x01
490409    // '-------------------------------------'
490410    // |
490411    // Enable interrupt when packet
490412    // received
490413    //
490414    out_8 ((io + NE2K_IMR), 0x01);
490415    //
490416    // Transmit configuration register (TCR)
490417    // .-------------------------------.
490418    // | - | - | - |OFST|ATD|LB1|LB0|CRC|
490419    // |-------------------------------|
490420    // | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x00
490421    // '-------------------------------'
490422    //
490423    // Normal operation.
490424    //
490425    out_8 ((io + NE2K_TCR), 0x00);
490426    //
490427    return (0);
490428  }
```

94.4.24 kernel/driver/nic/ne2k/ne2k_rx.c

Si veda la sezione 93.16.

```
500001  #include <kernel/driver/pci.h>
500002  #include <kernel/net.h>
500003  #include <kernel/driver/nic/ne2k.h>
500004  #include <kernel/ibm_i386.h>
500005  #include <errno.h>
500006  #include <kernel/lib_k.h>
500007  #include <kernel/lib_s.h>
500008  //-------------------------------------------------------
500009  #define DEBUG 0
500010  //-------------------------------------------------------
500011  int
500012  ne2k_rx (uintptr_t io)
500013  {
500014    int i;
500015    int bytes;
500016    int curr;
500017    int bnry;
500018    int next;
500019    int frame_status;
500020    int frame_size;
500021    int status;
500022    int n = net_index_eth (0, NULL, io);
500023    net_buffer_eth_t *buffer;
500024    //
500025    // Verify to have found a valid Ethernet device.
500026    //
500027    if (n < 0)
500028      {
500029        errset (ENODEV);
500030        return (-1);
500031      }
500032    //
500033    // Command register (CR)
500034    // .-----------------------------.
500035    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
500036    // |-----------------------------|
500037    // | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0x62
500038    // '-----------------------------'
500039    // \_____/ \_________/ |
500040    // | | START
500041    // | |
500042    // | Abort/complete remote DMA
500043    // |
500044    // Register page 1
500045    //
500046    out_8 ((io + NE2K_CR), 0x62);
500047    //
500048    // Get the current position.
500049    //
500050    curr = in_8 (io + NE2K_CURR);
500051    //
500052    // Command register (CR)
500053    // .-----------------------------.
500054    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
500055    // |-----------------------------|
```

```
500056    // | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0x22
500057    // '-------------------------------'
500058    // \_____/ \_________/ |
500059    // | | START
500060    // | |
500061    // | Abort/complete remote DMA
500062    // |
500063    // Register page 0
500064    //
500065    out_8 ((io + NE2K_CR), 0x22);
500066    //
500067    // Get the boundary.
500068    //
500069    bnry = in_8 (io + NE2K_BNRY);
500070    //
500071    // -------------------------------------------------
500072    // The function is run because at least a frame was
500073    // received:
500074    // if index 'bnry' and index 'curr' are the same,
500075    // all the receive
500076    // ring buffer is to be copied.
500077    // -------------------------------------------------
500078    //
500079    // Get all the frames ready from the internal
500080    // buffer.
500081    //
500082    while (1)
500083      {
500084        //
500085        // Find a place inside the frame table.
500086        //
500087        buffer = net_buffer_eth (n);
500088        //
500089        // Check to have a valid buffer pointer.
500090        //
500091        if (buffer == NULL)
500092          {
500093            errset (errno);
500094            return (-1);
500095          }
500096        //
500097        // First read 4 bytes starting from 'bnry'.
500098        //
500099        out_8 ((io + NE2K_RBCR0), 4);
500100        out_8 ((io + NE2K_RBCR1), 0);
500101        //
500102        // Set the remote DMA address to bnry.
500103        //
500104        out_8 ((io + NE2K_RSAR0), 0x00);  // Must be
500105        // zero.
500106        out_8 ((io + NE2K_RSAR1), bnry);
500107        //
500108        // Command register (CR)
500109        // .-------------------------------.
500110        // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
500111        // |-------------------------------|
500112        // | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
500113        // '-------------------------------'
500114        // \_____/ \_________/ |
500115        // | | START
500116        // | |
500117        // | Read
500118        // |
500119        // Register page 0
500120        //
500121        out_8 (io + NE2K_CR, 0x0A);
500122        //
500123        // Frame status
500124        //
500125        frame_status = in_8 (io + NE2K_DATA);
500126        //
500127        // Next frame.
500128        //
500129        next = in_8 (io + NE2K_DATA);
500130        //
500131        // Frame size low.
500132        //
500133        frame_size = in_8 (io + NE2K_DATA);
500134        //
500135        // Frame size high
500136        //
500137        frame_size += (in_8 (io + NE2K_DATA) * 256);
500138        //
500139        // Interrupt status register (ISR)
500140        // .-------------------------------.
500141        // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
500142        // |-------------------------------|
```



```
500143        // | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
500144        // '-------------------------------'
500145        // |
500146        // Remote DMA complete
500147        //
500148        // Verify to have finished with DMA transfer.
500149        //
500150        status = ne2k_isr_expect (io, 0x40);
500151        if (status)
500152          {
500153            errset (errno);
500154            return (-1);
500155          }
500156        //
500157        // Now read again all the frame plus header (the
500158        // initial 4 bytes).
500159        //
500160        buffer->clock = k_clock ();
500161        buffer->size = frame_size - 4;
500162        //
500163        if (DEBUG)
500164          {
500165            k_printf
500166              ("0x%02x[BNRY=0x%02x "
500167               "CURR=0x%02x]0x%02x size=%i\n",
500168               NE2K_RX_START, bnry, curr, NE2K_RX_STOP,
500169               frame_size);
500170          }
500171        //
500172        if (next == bnry)
500173          {
500174            k_printf
500175              ("[%s] next==bnry but should "
500176               "not happen!\n", __func__);
500177            errset (E_DRIVER_FAULT);
500178            return (-1);
500179          }
500180        //
500181        if (next > bnry)
500182          {
500183            bytes = frame_size;
500184          }
500185        //
500186        if (next < bnry)
500187          {
500188            //
500189            // Read up to the bottom.
500190            //
500191            bytes = ((NE2K_RX_STOP - bnry) * 256);
500192            bytes = min (bytes, frame_size);
500193          }
500194        //
500195        // Read frame content: first part.
500196        //
500197        out_8 ((io + NE2K_RBCR0), bytes & 0xFF);
500198        out_8 ((io + NE2K_RBCR1), bytes >> 8);
500199        //
500200        out_8 ((io + NE2K_RSAR0), 0);     // MUST be
500201        // zero. :-(
500202        out_8 ((io + NE2K_RSAR1), bnry);
500203        //
500204        // Command register (CR)
500205        // .-------------------------------.
500206        // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
500207        // |-------------------------------|
500208        // | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
500209        // '-------------------------------'
500210        // \_____/ \_________/ |
500211        // | | START
500212        // | |
500213        // | Read
500214        // |
500215        // Register page 0
500216        //
500217        out_8 (io + NE2K_CR, 0x0A);
500218        //
500219        // Jump the first four bytes (no way to start
500220        // after
500221        // the page start).
500222        //
500223        in_8 (io + NE2K_DATA);
500224        in_8 (io + NE2K_DATA);
500225        in_8 (io + NE2K_DATA);
500226        in_8 (io + NE2K_DATA);
500227        bytes -= 4;
500228        //
500229        // Get the frame data.
```

```
500230        //
500231        i = 0;
500232        for (; bytes > 0; i++, bytes--)
500233          {
500234            buffer->frame.octet[i] = in_8 (io + NE2K_DATA);
500235          }
500236        //
500237        // Interrupt status register (ISR)
500238        // .-------------------------------.
500239        // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
500240        // |-------------------------------|
500241        // | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
500242        // '-------------------------------'
500243        // |
500244        // Remote DMA complete
500245        //
500246        // Verify to have finished with DMA transfer.
500247        //
500248        status = ne2k_isr_expect (io, 0x40);
500249        if (status)
500250          {
500251            errset (errno);
500252            return (-1);
500253          }
500254        //
500255        if (next < bnry)
500256          {
500257            //
500258            // There might be a second part to read.
500259            //
500260            bytes =
500261              frame_size - ((NE2K_RX_STOP - bnry) * 256);
500262          }
500263        //
500264        if (bytes > 0)
500265          {
500266            //
500267            out_8 ((io + NE2K_RBCR0), bytes & 0xFF);
500268            out_8 ((io + NE2K_RBCR1), bytes >> 8);
500269            //
500270            out_8 ((io + NE2K_RSAR0), 0);
500271            out_8 ((io + NE2K_RSAR1), NE2K_RX_START);
500272            //
500273            // Command register (CR)
500274            // .-------------------------------.
500275            // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
500276            // |-------------------------------|
500277            // | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
500278            // '-------------------------------'
500279            // \_____/ \_________/ |
500280            // | | START
500281            // | |
500282            // | Read
500283            // |
500284            // Register page 0
500285            //
500286            out_8 (io + NE2K_CR, 0x0A);
500287            //
500288            for (; bytes > 0; i++, bytes--)
500289              {
500290                buffer->frame.octet[i] =
500291                  in_8 (io + NE2K_DATA);
500292              }
500293            //
500294            // Interrupt status register (ISR)
500295            // .-------------------------------.
500296            // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
500297            // |-------------------------------|
500298            // | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
500299            // '-------------------------------'
500300            // |
500301            // Remote DMA complete
500302            //
500303            // Verify to have finished with DMA
500304            // transfer.
500305            //
500306            status = ne2k_isr_expect (io, 0x40);
500307            if (status)
500308              {
500309                errset (errno);
500310                return (-1);
500311              }
500312          }
500313        //
500314        // Update BNRY.
500315        //
500316        bnry = next;
```



```
500317        out_8 (io + NE2K_BNRY, bnry);
500318        //
500319        // If the new bnry is equal to curr, the loop is
500320        // finished.
500321        //
500322        if (bnry == curr)
500323          {
500324            //
500325            // finish.
500326            //
500327            return (0);
500328          }
500329      }
500330  }
```

94.4.25 kernel/driver/nic/ne2k/ne2k_rx_reset.c

Si veda la sezione 93.16.

```
510001  #include <kernel/driver/pci.h>
510002  #include <kernel/driver/nic/ne2k.h>
510003  #include <kernel/ibm_i386.h>
510004  #include <errno.h>
510005  #include <kernel/lib_k.h>
510006  #include <kernel/lib_s.h>
510007  //-------------------------------------------------------
510008  #define DEBUG 1
510009  //-------------------------------------------------------
510010  int
510011  ne2k_rx_reset (uintptr_t io)
510012  {
510013    int status;
510014    //
510015    // Command register (CR)
510016    // .-------------------------------.
510017    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
510018    // |-------------------------------|
510019    // | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
510020    // '-------------------------------'
510021    // \_____/ \_________/ |
510022    // | | STOP
510023    // | |
510024    // | Abort/complete remote DMA
510025    // |
510026    // Register page 0
510027    //
510028    out_8 ((io + NE2K_CR), 0x21);
510029    //
510030    // Interrupt status register (ISR)
510031    // .-------------------------------.
510032    // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
510033    // |-------------------------------|
510034    // | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x80
510035    // '-------------------------------'
510036    // |
510037    // Reset
510038    //
510039    status = ne2k_isr_expect (io, 0x80);
510040    if (status)
510041      {
510042        errset (errno);
510043        return (-1);
510044      }
510045    //
510046    //
510047    //
510048    out_8 ((io + NE2K_RBCR0), 0);
510049    out_8 ((io + NE2K_RBCR1), 0);
510050    //
510051    // Command register (CR)
510052    // .-------------------------------.
510053    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
510054    // |-------------------------------|
510055    // | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0x22
510056    // '-------------------------------'
510057    // \_____/ \_________/ |
510058    // | | START
510059    // | |
510060    // | Abort/complete remote DMA
510061    // |
510062    // Register page 0
510063    //
510064    out_8 (io + NE2K_CR, 0x22);
510065    //
510066    return (0);
510067  }
```

94.4.26 kernel/driver/nic/ne2k/ne2k_tx.c

Si veda la sezione 93.16.

```
520001  #include <kernel/driver/pci.h>
520002  #include <kernel/driver/nic/ne2k.h>
520003  #include <kernel/ibm_i386.h>
520004  #include <errno.h>
520005  #include <kernel/lib_k.h>
520006  #include <kernel/lib_s.h>
520007  //-------------------------------------------------------
520008  int
520009  ne2k_tx (uintptr_t io, void *buffer, size_t size)
520010  {
520011    int i;
520012    int status;
520013    uint8_t *b = buffer;
520014    //
520015    // Read the command register to see if the NIC is
520016    // transmitting.
520017    // The value 0x26 tells that the NIC is
520018    // transmitting.
520019    //
520020    // Command register (CR)
520021    // .-------------------------------.
520022    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
520023    // |-------------------------------|
520024    // | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0x26
520025    // '-------------------------------'
520026    // \_____/ \_________/ | |
520027    // | | | Start
520028    // | | Transmit frame
520029    // | Abort/complete
520030    // | remote DMA
520031    // Register
520032    // page 0
520033    //
520034    status = in_8 (io + NE2K_CR);
520035    if (status == 0x26)
520036      {
520037        errset (EBUSY);
520038        return (-1);
520039      }
520040    //
520041    // Set up the frame size: the size is split into
520042    // RBCR0 and RBCR1
520043    // registers.
520044    //
520045    out_8 ((io + NE2K_RBCR0), (size & 0xFF));
520046    out_8 ((io + NE2K_RBCR1), (size >> 8));
520047    //
520048    // Set the remote DMA address.
520049    //
520050    out_8 ((io + NE2K_RSAR0), 0x00);      // Must be
520051    // zero.
520052    out_8 ((io + NE2K_RSAR1), NE2K_TX_BUFFER);
520053    //
520054    // Command register (CR)
520055    // .-------------------------------.
520056    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
520057    // |-------------------------------|
520058    // | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0x12
520059    // '-------------------------------'
520060    // \_____/ \_________/ |
520061    // | | Start
520062    // | |
520063    // | Write
520064    // Register
520065    // page 0
520066    //
520067    out_8 ((io + NE2K_CR), 0x12);
520068    //
520069    // Write to the data port all the frame.
520070    //
520071    for (i = 0; i < size; i++)
520072      {
520073        out_8 ((io + NE2K_DATA), b[i]);
520074      }
520075    //
520076    // Interrupt status register (ISR)
520077    // .-------------------------------.
520078    // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
520079    // |-------------------------------|
520080    // | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
520081    // '-------------------------------'
520082    // |
520083    // Remote DMA complete
520084    //
520085    // Verify to have finished with DMA transfer.
520086    //
520087    status = ne2k_isr_expect (io, 0x40);
520088    if (status)
520089      {
520090        errset (errno);
520091        return (-1);
520092      }
520093    //
520094    // Set transmit page start, to the transmit buffer.
520095    //
520096    out_8 (io + NE2K_TPSR, NE2K_TX_BUFFER);
520097    //
520098    // Set transmit byte count (frame size).
520099    //
520100    out_8 ((io + NE2K_TBCR0), (size & 0xFF));
520101    out_8 ((io + NE2K_TBCR1), (size >> 8));
520102    //
520103    // Command register (CR)
520104    // .-------------------------------.
520105    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
520106    // |-------------------------------|
520107    // | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0x26
520108    // '-------------------------------'
520109    // \_____/ \_________/ | |
520110    // | | | Start
520111    // | | Transmit frame
520112    // | Abort/complete remote DMA
520113    // Register
520114    // page 0
520115    //
520116    // Send frame!
520117    //
520118    out_8 ((io + NE2K_CR), 0x26);
520119    //
520120    // Interrupt status register (ISR)
520121    // .-------------------------------.
520122    // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
520123    // |-------------------------------|
520124    // | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
520125    // '-------------------------------'
520126    // | |
520127    // | Frame transmitted with no errors
520128    // Transmit error
520129    //
520130    // Wait the end of transmission: might get a good
520131    // transmission
520132    // report, or an error transmission report.
520133    //
520134    status = ne2k_isr_expect (io, 0x0A);
520135    if (status)
520136      {
520137        errset (errno);
520138        return (-1);
520139      }
520140    //
520141    // Transmit status (TSR)
520142    // .-------------------------------.
520143    // |OWC|CDH| FU|CRS|ABT|COL| - |PTX|
520144    // |-------------------------------|
520145    // | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0x38
520146    // '-------------------------------'
520147    // | | |
520148    // | | Transmit aborted
520149    // | Carrier sense lost
520150    // FIFO underrun
520151    //
520152    // Check if there was an error, during transmission.
520153    //
520154    status = in_8 (io + NE2K_TSR);
520155    if (status & 0x38)
520156      {
520157        errset (EIO);
520158        return (-1);
520159      }
520160    //
520161    // Done.
520162    //
520163    return (0);
520164  //
520165  }
```


94.4.27 kernel/driver/pci.h

Si veda la sezione 93.19.

```
530001  #ifndef _KERNEL_DRIVER_PCI_H
530002  #define _KERNEL_DRIVER_PCI_H    1
```

```
530003 //-------------------------------------------------------
530004 #include <stdint.h>
530005 #include <sys/types.h>
530006 //-------------------------------------------------------
530007 //
530008 #define PCI_MAX_DEVICES                 8
530009 #define PCI_MAX_BUSES                 256       // Fixed.
530010 #define PCI_MAX_SLOTS                  32       // Fixed.
530011 //
530012 #define PCI_CONFIG_ADDRESS          0x0CF8
530013 #define PCI_CONFIG_DATA             0x0CFC
530014 //
530015 // CONFIG_ADDRESS register structure.
530016 //
530017 typedef union
530018 {
530019   uint32_t selector;
530020   struct
530021   {
530022     uint32_t zero:2,
530023       reg:6,
530024       function:3, slot:5, bus:8, reserved:7, enable:1;
530025   };
530026 } pci_address_t;
530027 //
530028 // CONFIG_DATA register structures.
530029 //
530030 typedef union
530031 {
530032   uint32_t r[16];
530033   struct
530034   {
530035     struct
530036     {
530037       uint32_t vendor_id:16, device_id:16;
530038       //
530039       uint32_t command:16, status:16;
530040       //
530041       uint32_t revision_id:8,
530042         prog_if:8, subclass:8, class_code:8;
530043       //
530044       uint32_t cache_line_size:8,
530045         latency_timer:8,
530046         header_type:7, multi_function:1, bist:8;
530047       //
530048       uint32_t bar0;
530049       uint32_t bar1;
530050       uint32_t bar2;
530051       uint32_t bar3;
530052       uint32_t bar4;
530053       uint32_t bar5;
530054       uint32_t cardbus_cis_pointer;
530055       uint32_t expansion_rom_base_address;
530056       //
530057       uint32_t subsystem_vendor_id:16, subsystem_id:16;
530058       //
530059       uint32_t capabilities_pointer:8, reserved_1:24;
530060       //
530061       uint32_t reserved_2;
530062       //
530063       uint32_t interrupt_line:8,
530064         interrupt_pin:8, min_grant:8, max_latency:8;
530065     };
530066   };
530067 } pci_header_type_00_t;
530068 //
530069 //-------------------------------------------------------
530070 //
530071 // PCI table row.
530072 //
530073 typedef struct
530074 {
530075   unsigned char bus;
530076   unsigned char slot;
530077   unsigned short int vendor_id;
530078   unsigned short int device_id;
530079   unsigned char class_code;
530080   unsigned char subclass;
530081   unsigned char prog_if;
530082   uintptr_t base_io;
530083   unsigned char irq;
530084 } pci_t;
530085 //
530086 extern pci_t pci_table[PCI_MAX_DEVICES];
530087 //
530088 //-------------------------------------------------------
530089 void pci_init (void);
530090 //-------------------------------------------------------
530091 #endif
```


94.4.28 kernel/driver/pci/pci_init.c

Si veda la sezione 93.19.

```
540001 #include <kernel/driver/pci.h>
540002 #include <kernel/ibm_i386.h>
540003 #include <errno.h>
540004 //-------------------------------------------------------
540005 extern pci_t pci_table[PCI_MAX_DEVICES];
540006 //-------------------------------------------------------
540007 void
540008 pci_init (void)
540009 {
540010   pci_header_type_00_t pci;
540011   pci_address_t pci_addr;
540012   //
540013   int t;        // PCI table index.
540014   int b;        // PCI bus index.
540015   int s;        // PCI slot index.
540016   int r;        // PCI header register index.
540017   //
540018   // Reset the PCI table.
540019   //
540020   for (t = 0; t < PCI_MAX_DEVICES; t++)
540021     {
540022       pci_table[t].bus = 0;
540023       pci_table[t].slot = 0;
540024       pci_table[t].vendor_id = 0;
540025       pci_table[t].device_id = 0;
540026       pci_table[t].class_code = 0;
540027       pci_table[t].subclass = 0;
540028       pci_table[t].prog_if = 0;
540029       pci_table[t].base_io = 0;
540030       pci_table[t].irq = 0;
540031     }
540032   //
540033   // Scan PCI buses and slots.
540034   //
540035   t = 0;
540036   //
540037   for (b = 0; b < PCI_MAX_BUSES && t < PCI_MAX_DEVICES; b++)
540038     {
540039       //
540040       // Will not check multi functions devices (we
540041       // are shure that
540042       // we don't have them).
540043       //
540044       for (s = 0;
540045            s < PCI_MAX_SLOTS && t < PCI_MAX_DEVICES; s++)
540046         {
540047           pci_addr.selector = 0;
540048           pci_addr.enable = 1;
540049           pci_addr.bus = b;
540050           pci_addr.slot = s;
540051           //
540052           pci_addr.reg = 0;
540053           out_32 (PCI_CONFIG_ADDRESS, pci_addr.selector);
540054           pci.r[0] = in_32 (PCI_CONFIG_DATA);
540055           //
540056           if (pci.r[0] == 0xFFFFFFFF)
540057             {
540058               //
540059               // There is no such bus:slot
540060               // combination!
540061               //
540062               continue;
540063             }
540064           else
540065             {
540066               for (r = 1; r < 16; r++)
540067                 {
540068                   pci_addr.reg = r;
540069                   out_32 (PCI_CONFIG_ADDRESS,
540070                           pci_addr.selector);
540071                   pci.r[r] = in_32 (PCI_CONFIG_DATA);
540072                 }
540073             }
540074           //
540075           // We consider only PCI header type 0x00!
540076           //
540077           if (pci.header_type != 0)
540078             {
540079               continue;
540080             }
```

```
540081            //
540082            // We do not consider PCI bridge devices!
540083            //
540084            if (pci.class_code == 0x06)
540085              {
540086                continue;
540087              }
540088            //
540089            // Save the device inside the PCI table.
540090            //
540091            pci_table[t].bus = b;
540092            pci_table[t].slot = s;
540093            pci_table[t].vendor_id = pci.vendor_id;
540094            pci_table[t].device_id = pci.device_id;
540095            pci_table[t].class_code = pci.class_code;
540096            pci_table[t].subclass = pci.subclass;
540097            pci_table[t].prog_if = pci.prog_if;
540098            pci_table[t].base_io = pci.bar0 & 0xFFFFFFFC;
540099            pci_table[t].irq = pci.interrupt_line;
540100            //
540101            k_printf ("[%s]: %04x:%04x io=%04x irq=%i\n",
540102                      __func__, pci_table[t].vendor_id,
540103                      pci_table[t].device_id,
540104                      pci_table[t].base_io, pci_table[t].irq);
540105            //
540106            // Next PCI table row.
540107            //
540108            t++;
540109          }
540110      }
540111  }
```

94.4.29 kernel/driver/pci/pci_public.c

«

Si veda la sezione 93.19.

```
550001  #include <kernel/driver/pci.h>
550002  //-------------------------------------------------------
550003  pci_t pci_table[PCI_MAX_DEVICES];
550004  //-------------------------------------------------------
```

94.4.30 kernel/driver/screen.h

«

Si veda la sezione 93.22.

```
560001  #ifndef _KERNEL_DRIVER_SCREEN_H
560002  #define _KERNEL_DRIVER_SCREEN_H 1
560003  //-------------------------------------------------------
560004  #include <restrict.h>
560005  #include <stdint.h>
560006  //-------------------------------------------------------
560007  // Virtual consoles data and VGA references.
560008  //
560009  #define SCREEN_MAX     4
560010  #define SCREEN_ROWS    25
560011  #define SCREEN_COLS    80
560012  #define SCREEN_CELLS  (SCREEN_ROWS * SCREEN_COLS)
560013  //
560014  #define VGA_ATTR       0x07
560015  #define VGA_ADDR       0xB8000
560016  #define VGA_CELL      ((uint16_t *) VGA_ADDR)
560017  //
560018  typedef struct
560019  {
560020    uint16_t cell[SCREEN_CELLS];  // [1]
560021    int position;
560022  } screen_t;
560023  //
560024  // [1] Every character on the screen needs another
560025  //     attribute byte.
560026  //
560027  //-------------------------------------------------------
560028  extern int screen_active;
560029  extern screen_t screen_table[];
560030  //-------------------------------------------------------
560031  int screen_clear (screen_t * screen);
560032  screen_t *screen_current (void);
560033  void screen_init (void);
560034  int screen_new_line (screen_t * screen);
560035  int screen_number (screen_t * screen);
560036  screen_t *screen_pointer (int scrn);
560037  int screen_putc (screen_t * screen, int c);
560038  int screen_scroll (screen_t * screen);
560039  int screen_select (screen_t * screen);
560040  void screen_update (screen_t * screen);
560041  //-------------------------------------------------------
560042  #define screen_cell(c, attrib) \
560043      ((uint16_t) c \
560044       | ((((uint16_t) attrib) << 8) & 0xFF00))
560045  //-------------------------------------------------------
560046  #endif
```


94.4.31 kernel/driver/screen/screen_clear.c

«

Si veda la sezione 93.22.

```
570001  #include <kernel/driver/screen.h>
570002  #include <errno.h>
570003  //-------------------------------------------------------
570004  int
570005  screen_clear (screen_t * screen)
570006  {
570007    int j;
570008    //
570009    // Check argument.
570010    //
570011    if (screen == NULL)
570012      {
570013        errset (EINVAL);
570014        return (-1);
570015      }
570016    //
570017    // Clear the virtual screen.
570018    //
570019    for (j = 0; j < SCREEN_CELLS; j++)
570020      {
570021        screen->cell[j] = screen_cell (' ', VGA_ATTR);
570022      }
570023    //
570024    // Place the cursor at the top.
570025    //
570026    screen->position = 0;
570027    //
570028    // Update the screen if it is the active one.
570029    //
570030    screen_update (screen);
570031    //
570032    // Ok.
570033    //
570034    return (0);
570035  }
```

94.4.32 kernel/driver/screen/screen_current.c

«

Si veda la sezione 93.22.

```
580001  #include <kernel/driver/screen.h>
580002  //-------------------------------------------------------
580003  screen_t *
580004  screen_current (void)
580005  {
580006    if (screen_active >= 0 && screen_active < SCREEN_MAX)
580007      {
580008        return &screen_table[screen_active];
580009      }
580010    else
580011      {
580012        return &screen_table[0];
580013      }
580014  }
```

94.4.33 kernel/driver/screen/screen_init.c

«

Si veda la sezione 93.22.

```
590001  #include <kernel/driver/screen.h>
590002  //-------------------------------------------------------
590003  void
590004  screen_init (void)
590005  {
590006    int i;
590007    int j;
590008    //
590009    // Reset and clear all virtual consoles.
590010    //
590011    for (i = 0; i < SCREEN_MAX; i++)
590012      {
590013        //
590014        // Reset position.
590015        //
590016        screen_table[i].position = 0;
590017        //
590018        for (j = 0; j < SCREEN_CELLS; j++)
590019          {
```

```
590020            screen_table[i].cell[j] =
590021              screen_cell (' ', VGA_ATTR);
590022          }
590023      }
590024    //
590025    // Select the first screen.
590026    //
590027    screen_active = 0;
590028  }
```

94.4.34 kernel/driver/screen/screen_new_line.c

«

Si veda la sezione 93.22.

```
600001  #include <kernel/driver/screen.h>
600002  #include <errno.h>
600003  //-----------------------------------------------------------
600004  int
600005  screen_new_line (screen_t * screen)
600006  {
600007    int row;
600008    //
600009    // Check argument.
600010    //
600011    if (screen == NULL)
600012      {
600013        errset (EINVAL);
600014        return (-1);
600015      }
600016    //
600017    // Find row position on screen.
600018    //
600019    row = (screen->position / SCREEN_COLS);
600020    //
600021    // We want to go one row down.
600022    //
600023    row++;
600024    //
600025    // Scroll the screen if necessary.
600026    //
600027    for (; row >= SCREEN_ROWS; row--)
600028      {
600029        screen_scroll (screen);
600030      }
600031    //
600032    // Reset position at the beginning of the line.
600033    //
600034    screen->position = row * SCREEN_COLS;
600035    //
600036    // Update the video if it is the current one. This
600037    // is necessary to
600038    // update the cursor position, if the original
600039    // column was not zero.
600040    //
600041    screen_update (screen);
600042    //
600043    // Ok.
600044    //
600045    return (0);
600046  }
```

94.4.35 kernel/driver/screen/screen_number.c

«

Si veda la sezione 93.22.

```
610001  #include <kernel/driver/screen.h>
610002  #include <stddef.h>
610003  #include <errno.h>
610004  #include <kernel/lib_k.h>
610005  //-----------------------------------------------------------
610006  int
610007  screen_number (screen_t * screen)
610008  {
610009    ptrdiff_t distance;
610010    int n;
610011    //
610012    if (screen == NULL)
610013      {
610014        errset (EINVAL);
610015        return (-1);
610016      }
610017    //
610018    distance = (void *) screen - (void *) &screen_table[0];
610019    //
610020    n = (distance % (sizeof (screen_t)));
610021    //
610022    if (n != 0)
```



```
610023      {
610024        errset (EINVAL);  // Invalid pointer placement.
610025        return (-1);
610026      }
610027    //
610028    n = (distance / (sizeof (screen_t)));
610029    //
610030    if (n < 0 || n > SCREEN_MAX)
610031      {
610032        errset (EINVAL);  // Pointer outside the screen
610033        // table.
610034        return (-1);
610035      }
610036    //
610037    // If we are here, variable 'n' holds the right
610038    // screen number.
610039    //
610040    return (n);
610041  }
```

94.4.36 kernel/driver/screen/screen_pointer.c

«

Si veda la sezione 93.22.

```
620001  #include <kernel/driver/screen.h>
620002  #include <errno.h>
620003  //-----------------------------------------------------------
620004  screen_t *
620005  screen_pointer (int scrn)
620006  {
620007    if (scrn >= 0 && scrn < SCREEN_MAX)
620008      {
620009        return &screen_table[scrn];
620010      }
620011    else
620012      {
620013        errset (EINVAL);
620014        return (NULL);
620015      }
620016  }
```

94.4.37 kernel/driver/screen/screen_public.c

«

Si veda la sezione 93.22.

```
630001  #include <kernel/driver/screen.h>
630002  #include <errno.h>
630003  //-----------------------------------------------------------
630004  int screen_active;
630005  screen_t screen_table[SCREEN_MAX];
```

94.4.38 kernel/driver/screen/screen_putc.c

«

Si veda la sezione 93.22.

```
640001  #include <kernel/driver/screen.h>
640002  #include <errno.h>
640003  //-----------------------------------------------------------
640004  int
640005  screen_putc (screen_t * screen, int c)
640006  {
640007    int row;
640008    int col;
640009    //
640010    if (screen == NULL)
640011      {
640012        errset (EINVAL);
640013        return (-1);
640014      }
640015    //
640016    // Find row-col position on screen.
640017    //
640018    row = (screen->position / SCREEN_COLS);
640019    col = (screen->position - (row * SCREEN_COLS));
640020    //
640021    //
640022    //
640023    if (c == '\n' || c == '\r')
640024      {
640025        screen_new_line (screen);
640026        return (0);
640027      }
640028    else if (c == '\b')
640029      {
640030        screen->position--;
640031        if (screen->position < 0)
640032          {
```

```
640033            screen->position = 0;
640034          }
640035        screen_update (screen);
640036        return (0);
640037      }
640038    else if (screen->position == (SCREEN_CELLS - 1))
640039      {
640040        //
640041        // It is not a control character and we are
640042        // already at the
640043        // last cell of the last row.
640044        //
640045        screen_scroll (screen);
640046      }
640047    //
640048    // If we are here, it is not a control character.
640049    // So: print it.
640050    //
640051    screen->cell[screen->position] =
640052      screen_cell (c, VGA_ATTR);
640053    screen->position++;
640054    screen_update (screen);
640055    //
640056    return (0);
640057  }
```

94.4.39 kernel/driver/screen/screen_scroll.c

«

Si veda la sezione 93.22.

```
650001  #include <kernel/driver/screen.h>
650002  #include <errno.h>
650003  //-----------------------------------------------------------
650004  int
650005  screen_scroll (screen_t * screen)
650006  {
650007    int a;        // screen[].cell[] index.
650008    int b;        // screen[].cell[] index
650009    //
650010    // Check argument.
650011    //
650012    if (screen == NULL)
650013      {
650014        errset (EINVAL);
650015        return (-1);
650016      }
650017    //
650018    // Move up a line.
650019    //
650020    for (a = 0, b = SCREEN_COLS; b < SCREEN_CELLS; a++, b++)
650021      {
650022        screen->cell[a] = screen->cell[b];
650023      }
650024    //
650025    // Clear last screen line.
650026    //
650027    for (b = (SCREEN_CELLS - SCREEN_COLS);
650028         b < SCREEN_CELLS; b++)
650029      {
650030        screen->cell[b] = screen_cell (' ', VGA_ATTR);
650031      }
650032    //
650033    // Update position.
650034    //
650035    screen->position -= SCREEN_COLS;
650036    if (screen->position < 0)
650037      {
650038        screen->position = 0;
650039      }
650040    //
650041    // Update the video if it is the current one.
650042    //
650043    screen_update (screen);
650044    //
650045    // Ok.
650046    //
650047    return (0);
650048  }
```

94.4.40 kernel/driver/screen/screen_select.c

«

Si veda la sezione 93.22.

```
660001  #include <kernel/driver/screen.h>
660002  #include <errno.h>
660003  #include <kernel/ibm_i386.h>
660004  //-----------------------------------------------------------
```



```
660005  int
660006  screen_select (screen_t * screen)
660007  {
660008    int scrn;
660009    //
660010    if (screen == NULL)
660011      {
660012        errset (EINVAL);
660013        return (-1);
660014      }
660015    //
660016    // Get screen number.
660017    //
660018    scrn = screen_number (screen);
660019    if (scrn < 0)
660020      {
660021        errset (EINVAL);  // The screen pointer was
660022        // invalid.
660023        return (-1);
660024      }
660025    //
660026    // Set the current screen, update the screen memory
660027    // and put the cursor.
660028    //
660029    screen_active = scrn;
660030    //
660031    screen_update (screen);
660032    //
660033    // Ok.
660034    //
660035    return (0);
660036  }
```

94.4.41 kernel/driver/screen/screen_update.c

«

Si veda la sezione 93.22.

```
670001  #include <kernel/driver/screen.h>
670002  #include <kernel/ibm_i386.h>
670003  #include <stddef.h>
670004  //-----------------------------------------------------------
670005  void
670006  screen_update (screen_t * screen)
670007  {
670008    screen_t *screen_showing;
670009    int j;
670010    unsigned char position_high;
670011    unsigned char position_low;
670012    //
670013    // Check input: if it is the NULL pointer, or it is
670014    // not a valid
670015    // pointer, then select the current screen.
670016    //
670017    if ((screen == NULL) || (screen_number (screen) < 0))
670018      {
670019        screen = screen_current ();
670020      }
670021    //
670022    // Get current screen anyway.
670023    //
670024    screen_showing = screen_current ();
670025    //
670026    // Verify again to be in a valid screen.
670027    //
670028    if (screen_number (screen_showing) < 0)
670029      {
670030        return;
670031      }
670032    //
670033    // If the selected screen is also the current
670034    // screen, then
670035    // must update the content (otherwise there is
670036    // nothing to do).
670037    //
670038    if (screen_showing == screen)
670039      {
670040        //
670041        // Copy virtual screen to real screen memory.
670042        //
670043        for (j = 0; j < SCREEN_CELLS; j++)
670044          {
670045            VGA_CELL[j] = screen->cell[j];
670046          }
670047        //
670048        // Place the cursor.
670049        //
670050        position_high =
```

```
670051          (unsigned char) (screen->position >> 8);
670052        position_low = (unsigned char) (screen->position);
670053        //
670054        out_8 (0x3D4, 0x0E);
670055        out_8 (0x3D5, position_high);
670056        out_8 (0x3D4, 0x0F);
670057        out_8 (0x3D5, position_low);
670058      }
670059  }
```

94.4.42 kernel/driver/tty.h

«

Si veda la sezione 93.24.

```
680001  #ifndef _KERNEL_DRIVER_TTY_H
680002  #define _KERNEL_DRIVER_TTY_H  1
680003  //-------------------------------------------------------
680004  #include <stddef.h>
680005  #include <stdint.h>
680006  #include <stdio.h>
680007  #include <sys/types.h>
680008  #include <kernel/ibm_i386.h>
680009  #include <termios.h>
680010  //-------------------------------------------------------
680011  #define   TTYS_CONSOLE    4
680012  #define   TTYS_SERIAL     0
680013  #define   TTYS_TOTAL      (TTYS_CONSOLE + TTYS_SERIAL)
680014  //-------------------------------------------------------
680015  #define   TTY_INPUT_LINE_EDITING 0
680016  #define   TTY_INPUT_LINE_CLOSED  1
680017  //-------------------------------------------------------
680018  typedef struct
680019  {
680020    dev_t device;
680021    pid_t pgrp;   // Process group.
680022    struct termios attr;  // termios attributes.
680023    unsigned char status; // 0 = edit, 1 = end edit.
680024    char line[MAX_CANON]; // Canonical input line.
680025    int lpr;      // Input line position read.
680026    int lpw;      // Input line position write.
680027  } tty_t;
680028  //-------------------------------------------------------
680029  extern tty_t tty_table[TTYS_TOTAL];
680030  //-------------------------------------------------------
680031  tty_t *tty_reference (dev_t device);
680032  dev_t tty_console (dev_t device);
680033  int tty_read (dev_t device);
680034  void tty_write (dev_t device, int c);
680035  void tty_init (void);
680036  //-------------------------------------------------------
680037  #endif
```

94.4.43 kernel/driver/tty/tty_console.c

«

Si veda la sezione 93.24.

```
690001  #include <sys/os32.h>
690002  #include <kernel/driver/tty.h>
690003  #include <kernel/driver/screen.h>
690004  //-------------------------------------------------------
690005  dev_t
690006  tty_console (dev_t device)
690007  {
690008    static dev_t device_active = DEV_CONSOLE0;    // First
690009    // time.
690010    dev_t device_previous;
690011    screen_t *screen;
690012    //
690013    // Check if it required only the current device.
690014    //
690015    if (device == 0)
690016      {
690017        return (device_active);
690018      }
690019    //
690020    // Fix if the device is not valid.
690021    //
690022    if (device > DEV_CONSOLE3 || device < DEV_CONSOLE0)
690023      {
690024        device = DEV_CONSOLE0;
690025      }
690026    //
690027    // Update.
690028    //
690029    device_previous = device_active;
690030    device_active = device;
690031    //
```



```
690032    // Get screen pointer.
690033    //
690034    screen = screen_pointer ((int) (device_active & 0x00FF));
690035    //
690036    // Switch.
690037    //
690038    screen_select (screen);
690039    //
690040    // Return previous device value.
690041    //
690042    return (device_previous);
690043  }
```

94.4.44 kernel/driver/tty/tty_init.c

«

Si veda la sezione 93.24.

```
700001  #include <sys/os32.h>
700002  #include <kernel/driver/tty.h>
700003  #include <kernel/driver/screen.h>
700004  #include <termios.h>
700005  //-------------------------------------------------------
700006  void
700007  tty_init (void)
700008  {
700009    int page;     // console page.
700010    //
700011    // Console initialization: console pages correspond
700012    // to the first
700013    // terminal items.
700014    //
700015    for (page = 0; page < TTYS_CONSOLE; page++)
700016      {
700017        tty_table[page].device = DEV_CONSOLE0 + page;
700018        tty_table[page].pgrp = 0;
700019        tty_table[page].line[0] = 0;
700020        tty_table[page].lpr = 0;
700021        tty_table[page].lpw = 0;
700022        tty_table[page].status = TTY_INPUT_LINE_EDITING;
700023        //
700024        // Termios default configuration.
700025        //
700026        tty_table[page].attr.c_iflag = BRKINT | ICRNL;
700027        tty_table[page].attr.c_oflag = 0;
700028        tty_table[page].attr.c_cflag = 0;
700029        tty_table[page].attr.c_lflag =
700030          ECHO | ECHOE | ECHOK | ECHONL | ICANON | ISIG;
700031        //
700032        // VEOF == ASCII EOT
700033        //
700034        tty_table[page].attr.c_cc[VEOF] = 0x04;
700035        //
700036        // VEOL == undefined
700037        //
700038        tty_table[page].attr.c_cc[VEOL] = 0x00;
700039        //
700040        // VERASE == ASCII BS
700041        //
700042        tty_table[page].attr.c_cc[VERASE] = 0x08;
700043        //
700044        // VINTR == ASCII ETX
700045        //
700046        tty_table[page].attr.c_cc[VINTR] = 0x03;
700047        //
700048        // VKILL == undefined
700049        //
700050        tty_table[page].attr.c_cc[VKILL] = 0x00;
700051        //
700052        // VMIN == 0
700053        //
700054        tty_table[page].attr.c_cc[VMIN] = 0x00;
700055        //
700056        // VQUIT == ASCII FS
700057        //
700058        tty_table[page].attr.c_cc[VQUIT] = 0x1C;
700059        //
700060        // VSTART == undefined
700061        //
700062        tty_table[page].attr.c_cc[VSTART] = 0x00;
700063        //
700064        // VSUSP == undefined
700065        //
700066        tty_table[page].attr.c_cc[VSUSP] = 0x00;
700067        //
700068        // VTIME == 0
700069        //
700070        tty_table[page].attr.c_cc[VTIME] = 0x00;
```

```
700071      }
700072    //
700073    // Set video mode.
700074    //
700075    screen_init ();
700076    //
700077    // Select the first console.
700078    //
700079    tty_console (DEV_CONSOLE0);
700080    //
700081    // Nothing else to configure (only consoles are
700082    // available).
700083    //
700084    return;
700085  }
```

94.4.45 kernel/driver/tty/tty_public.c

Si veda la sezione 93.24.

```
710001  #include <kernel/driver/tty.h>
710002  //-------------------------------------------------------
710003  tty_t tty_table[TTYS_TOTAL];
```

94.4.46 kernel/driver/tty/tty_read.c

Si veda la sezione 93.24.

```
720001  #include <sys/os32.h>
720002  #include <kernel/driver/tty.h>
720003  #include <kernel/lib_k.h>
720004  //-------------------------------------------------------
720005  int
720006  tty_read (dev_t device)
720007  {
720008    tty_t *tty;
720009    int key;
720010    //
720011    tty = tty_reference (device);
720012    if (tty == NULL)
720013      {
720014        k_printf
720015          ("kernel alert: cannot find terminal device "
720016           "0x%08x!\n", (int) device);
720017        //
720018        return (-1);
720019      }
720020    //
720021    // Read from canonical input line, but only if it is
720022    // time to read.
720023    //
720024    if (tty->status == TTY_INPUT_LINE_CLOSED)
720025      {
720026        if (tty->lpr > tty->lpw)
720027          {
720028            //
720029            // There is nothing to read!
720030            // Reset input line.
720031            //
720032            tty->lpw = 0;
720033            tty->lpr = 0;
720034            tty->status = TTY_INPUT_LINE_EDITING;
720035            //
720036            return (-1);
720037          }
720038        //
720039        // Read the key.
720040        //
720041        key = tty->line[tty->lpr];
720042        //
720043        // Move up the read cursor.
720044        //
720045        tty->lpr++;
720046      }
720047    else
720048      {
720049        return (-1);
720050      }
720051    //
720052    // Return the key.
720053    //
720054    return (key);
720055
720056  }
```


94.4.47 kernel/driver/tty/tty_reference.c

Si veda la sezione 93.24.

```
730001  #include <kernel/driver/tty.h>
730002  //-------------------------------------------------------
730003  tty_t *
730004  tty_reference (dev_t device)
730005  {
730006    int t;        // Terminal index.
730007    //
730008    // If device is zero, a reference to the whole table
730009    // is returned.
730010    //
730011    if (device == 0)
730012      {
730013        return (tty_table);
730014      }
730015    //
730016    // Otherwise, a scan is made to find the selected
730017    // device.
730018    //
730019    for (t = 0; t < TTYS_TOTAL; t++)
730020      {
730021        if (tty_table[t].device == device)
730022          {
730023            //
730024            // Device found. Return the pointer.
730025            //
730026            return (&tty_table[t]);
730027          }
730028      }
730029    //
730030    // No device found!
730031    //
730032    return (NULL);
730033  }
```

94.4.48 kernel/driver/tty/tty_write.c

Si veda la sezione 93.24.

```
740001  #include <sys/os32.h>
740002  #include <kernel/driver/tty.h>
740003  #include <kernel/driver/screen.h>
740004  //-------------------------------------------------------
740005  void
740006  tty_write (dev_t device, int c)
740007  {
740008    screen_t *screen;
740009    //
740010    if ((device & 0xFF00) == (DEV_CONSOLE_MAJOR << 8))
740011      {
740012        //
740013        // Get screen pointer.
740014        //
740015        screen = screen_pointer ((int) (device & 0x00FF));
740016        //
740017        screen_putc (screen, c);
740018      }
740019  }
```

94.5 os32: «kernel/fs.h»

Si veda la sezione 93.6.

```
750001  #ifndef _KERNEL_FS_H
750002  #define _KERNEL_FS_H 1
750003  //-------------------------------------------------------
750004  #include <stdint.h>
750005  #include <sys/types.h>
750006  #include <sys/stat.h>
750007  #include <stdio.h>
750008  #include <limits.h>
750009  #include <kernel/memory.h>
750010  #include <sys/socket.h>
750011  #include <netinet/in.h>
750012  #include <netinet/tcp.h>
750013  #include <kernel/net/ip.h>
750014  #include <kernel/net/tcp.h>
750015  #include <sys/os32.h>
750016  //-------------------------------------------------------
750017  #define SB_MAX_INODE_BLOCKS    8        // 8*8192
750018                                          // inodes max.
750019  #define SB_MAX_ZONE_BLOCKS     8        // 8*8192
750020                                          // data-zones
750021                                          // max.
750022  #define SB_BLOCK_SIZE       1024        // Fixed for
```

```
750023                                  // Minix file
750024                                  // system.
750025  #define SB_MAX_ZONE_SIZE    4096        // log2 max is
750026                                  // 1.
750027  //---------------------------------------------------------
750028  //
750029  // blocks * (1024 * 8 / 16)
750030  //   = number of bits, divided 16.
750031  //
750032  #define SB_MAP_INODE_SIZE   (SB_MAX_INODE_BLOCKS*512)
750033  #define SB_MAP_ZONE_SIZE    (SB_MAX_ZONE_BLOCKS*512)
750034  //---------------------------------------------------------
750035  //
750036  // Number of zone pointers contained inside a zone,
750037  // used as an indirect inode list
750038  // (a pointer = 16 bits = 2 bytes).
750039  //
750040  #define INODE_MAX_INDIRECT_ZONES  (SB_MAX_ZONE_SIZE/2)
750041  //---------------------------------------------------------
750042  #define INODE_MAX_REFERENCES      0xFF
750043  //---------------------------------------------------------
750044  typedef uint16_t zno_t; // Zone number.
750045  //---------------------------------------------------------
750046  // The structured type 'inode_t' must be pre-declared
750047  // here, because the type sb_t, described before the
750048  // inode structure, has a member pointing to a type
750049  // 'inode_t'. So, must be declared previously the type
750050  // 'inode_t' as made of a type 'struct inode', then the
750051  // structure 'inode' can be described. But for a matter
750052  // of coherence, all other structured data declared
750053  // inside this file follow the same procedure.
750054  //
750055  typedef struct sb sb_t;
750056  typedef struct inode inode_t;
750057  typedef struct sock sock_t;
750058  typedef struct file file_t;
750059  typedef struct fd fd_t;
750060  typedef struct directory directory_t;
750061  //---------------------------------------------------------
750062  #define SB_MAX_SLOTS  16        // Handle max 16 file
750063                                  // systems.
750064
750065  struct sb
750066  {       // File system super block:
750067    uint16_t inodes;      // inodes available;
750068    uint16_t zones;       // zones available (disk
750069    // size);
750070    uint16_t map_inode_blocks;    // inode bit map
750071    // blocks;
750072    uint16_t map_zone_blocks;     // data-zone bit map
750073    // blocks;
750074    uint16_t first_data_zone;     // first data-zone;
750075    uint16_t log2_size_zone;      // log_2
750076    // (size_zone/block_size);
750077    uint32_t max_file_size;       // max file size in
750078    // bytes;
750079    uint16_t magic_number;        // file system magic
750080    // number.
750081    // ------------------------------------------------
750082    // Extra management data, not saved inside the file
750083    // system
750084    // super block.
750085    // ------------------------------------------------
750086    dev_t device; // FS device [3]
750087    inode_t *inode_mounted_on;    // [4]
750088    blksize_t blksize;    // Calculated zone size.
750089    int options;  // [5]
750090    uint16_t map_inode[SB_MAP_INODE_SIZE];
750091    uint16_t map_zone[SB_MAP_ZONE_SIZE];
750092    char changed;
750093  };
750094
750095  extern sb_t sb_table[SB_MAX_SLOTS];
750096  //
750097  // [3] the member 'device' must be kept at the same
750098  //     position, because it is used to calculate the
750099  //     super block header size, saved on disk.
750100  //
750101  // [4] If this pointer is not NULL, the super block is
750102  //     related to a device mounted on a directory. The
750103  //     inode of such directory is recorded here. Please
750104  //     note that it is type 'void *', instead of type
750105  //     'inode_t', because type 'inode_t' is declared
750106  //     after type 'sb_t'.
750107  //     Please note that the type 'sb_t' is declared
750108  //     before the type 'inode_t', but this member
750109  //     points to a type 'inode_t'.
```

```
750110  //     This is the reason because it was necessary to
750111  //     declare first the type 'inode_t' as made of
750112  //     'struct inode', to be described later. For
750113  //     coherence, all derived type made of structured
750114  //     data, are first declared as structure, and then,
750115  //     later, described.
750116  //
750117  // [5] Mount options can be only 'MOUNT_DEFAULT' or
750118  //     'MOUNT_RO', as defined inside file
750119  //     'lib/sys/os32.h'.
750120  //
750121  //---------------------------------------------------------
750122  #define INODE_MAX_SLOTS         (32 * OPEN_MAX)
750123  #define INODE_PIPE_BUFFER_SIZE  18      // (7 dir. + 2
750124                                  // ind.) * 2.
750125  //
750126  struct inode
750127  {       // Inode (32 byte total):
750128    uint16_t mode;        // file type and permissions;
750129    uint16_t uid; // user ID (16 bit);
750130    uint32_t size;        // file size in bytes;
750131    uint32_t time;        // file data modification
750132    // time;
750133    uint8_t gid;  // group ID (8 bit);
750134    uint8_t links;        // links to the inode;
750135    uint16_t direct[7];   // direct zones;
750136    uint16_t indirect1;   // indirect zones;
750137    uint16_t indirect2;   // double indirect zones.
750138    // ------------------------------------------------
750139    // Extra management data, not saved inside the disk
750140    // file system.
750141    // ------------------------------------------------
750142    sb_t *sb;     // Inode's super block. [7]
750143    ino_t ino;    // Inode number.
750144    sb_t *sb_attached;    // [8]
750145    blkcnt_t blkcnt;      // Rounded size/blksize.
750146    unsigned char references;     // Run time active
750147    // references.
750148    char changed:1,       // 1 == to be saved.
750149      pipe_dir:1; // 0 == read, 1 == write.
750150    unsigned char pipe_off_read;  // Pipe read offset.
750151    unsigned char pipe_off_write; // Pipe write offset
750152    unsigned char pipe_ref_read;  // Pipe read
750153    // references.
750154    unsigned char pipe_ref_write; // Pipe write
750155    // references
750156  };
750157
750158  extern inode_t inode_table[INODE_MAX_SLOTS];
750159  //
750160  // [7] the member 'sb' must be kept at the same
750161  //     position, because it is used to calculate the
750162  //     inode header size, saved on disk.
750163  //
750164  // [8] If the inode is a mount point for another
750165  //     device, the other super block pointer is saved
750166  //     inside 'sb_attached'.
750167  //
750168  //---------------------------------------------------------
750169  #define SOCK_MAX_SLOTS          64
750170  #define SOCK_MAX_QUEUE          (SOCK_MAX_SLOTS/4)
750171  //
750172  struct sock
750173  {
750174    int family;
750175    int type;
750176    int protocol;
750177    h_addr_t laddr;       // Local address, host byte
750178    // order.
750179    h_port_t lport;       // Local port, host byte
750180    // order.
750181    h_addr_t raddr;       // Remote address, host byte
750182    // order.
750183    h_port_t rport;       // Remote port, host byte
750184    // order.
750185    struct
750186    {
750187      clock_t clock[IP_MAX_PACKETS];      // [9]
750188    } read;
750189    uint8_t active:1,     // Is the socket used?
750190      unreach_net:1,      //
750191      unreach_host:1,     // ICMP unreachable status.
750192      unreach_prot:1,     //
750193      unreach_port:1;     //
750194    struct
750195    {
750196      uint16_t conn:4,    // Connection status.
```

```
750197        can_write:1,      // Can write to send_data[].
750198        can_read:1,       // Can read from *recv_index.
750199        can_send:1,       // Can send data.
750200        can_recv:1,       // Can receive data.
750201        send_closed:1,    // Closed send direction.
750202        recv_closed:1;    // Closed receive direction.
750203      //
750204      uint32_t lsq[16];   // Local sequence array.
750205      uint32_t lsq_ack;   // Expected acknowledge.
750206      uint32_t rsq[16];   // Remote sequence array.
750207      uint8_t lsqi:4,     // Local sequence array index.
750208        rsqi:4;   // Remote sequence array index.
750209      //
750210      clock_t clock;      // When was last send.
750211      //
750212      uint8_t send_data[TCP_MSS - sizeof (struct tcphdr)];
750213      size_t send_size;   // Size of 'send_data[]'
750214      // content.
750215      int send_flags;
750216      uint8_t recv_data[TCP_MAX_DATA_SIZE];       // Data
750217      // received.
750218      size_t recv_size;   // Size of 'recv_data[]'
750219      // content.
750220      uint8_t *recv_index;        // Read index inside
750221      // 'recv_data[]'.
750222      pid_t listen_pid;   // Process listening at local
750223      // port.
750224      int listen_max;     // Max connection requests.
750225      int listen_queue[SOCK_MAX_QUEUE];   // [10]
750226    } tcp;
750227  };
750228  //
750229  extern sock_t sock_table[SOCK_MAX_SLOTS];
750230  //
750231  // [9] The array 'read.clock[]' has the same size as
750232  //     the array as 'ip_tables[]', so that it can be
750233  //     saved, inside the former, the clock time of a
750234  //     packet read for the socket purposes.
750235  //     This is necessary to know if the packet was
750236  //     already managed inside the socket system, or
750237  //     it is new.
750238  //
750239  // [10] When a process listen o a local port, member
750240  //      'listen_pid' contains the pid number; member
750241  //      'listen_max' contains the max allowed
750242  //      connections that will be serviced; the array
750243  //      'listen_queue[]' will contain the file
750244  //      descriptors of established connections.
750245  //      If 'listen_queue[x]' is equal to -1, it means
750246  //      that there is no file descriptor there.
750247  //
750248  //-------------------------------------------------------
750249  #define FILE_MAX_SLOTS            (64 * OPEN_MAX)
750250
750251  struct file
750252  {
750253    int references;
750254    off_t offset; // File position.
750255    int oflags;   // Open mode: r/w/r+w [11]
750256    inode_t *inode;
750257    sock_t *sock;
750258  };
750259
750260  extern file_t file_table[FILE_MAX_SLOTS];
750261  //
750262  // [11] the member 'oflags' can get only O_RDONLY,
750263  //      O_WRONLY, O_RDWR, (from header 'fcntl.h')
750264  //      combined with OR binary operator.
750265  //
750266  //-------------------------------------------------------
750267  struct fd
750268  {
750269    int fl_flags; // File status flags and file
750270    // access modes. [12]
750271    int fd_flags; // File descriptor flags:
750272    // currently only FD_CLOEXEC.
750273    file_t *file; // Pointer to the file table.
750274  };
750275  //
750276  // [12] the member 'fl_flags' can get only O_RDONLY,
750277  //      O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_NOCTTY,
750278  //      O_TRUNC, O_APPEND and O_NONBLOCK
750279  //      (from header 'fcntl.h') combined with OR
750280  //      binary operator.
750281  //      Options like O_DSYNC, O_RSYNC and O_SYNC are
750282  //      not taken into consideration by os32.
750283  //
```



```
750284  //  Please notice that each process has its own 'fd'
750285  //  table, embedded inside the process table.
750286  //-------------------------------------------------------
750287  struct directory
750288  {       // Directory entry:
750289    uint16_t ino; // inode number;
750290    char name[NAME_MAX];  // file name.
750291  };
750292  //-------------------------------------------------------
750293  void fs_init (void);
750294  //-------------------------------------------------------
750295  int sb_inode_status (sb_t * sb, ino_t ino);
750296  sb_t *sb_mount (dev_t device, inode_t ** inode_mnt,
750297                  int options);
750298  void sb_print (void);
750299  sb_t *sb_reference (dev_t device);
750300  int sb_save (sb_t * sb);
750301  int sb_zone_status (sb_t * sb, zno_t zone);
750302  //-------------------------------------------------------
750303  zno_t zone_alloc (sb_t * sb);
750304  int zone_free (sb_t * sb, zno_t zone);
750305  void zone_print (sb_t * sb, zno_t zone);
750306  int zone_read (sb_t * sb, zno_t zone, void *buffer);
750307  int zone_write (sb_t * sb, zno_t zone, void *buffer);
750308  //-------------------------------------------------------
750309  inode_t *inode_alloc (dev_t device, mode_t mode,
750310                        uid_t uid, gid_t gid);
750311  int inode_check (inode_t * inode, mode_t type,
750312                   int perm, uid_t uid, gid_t gid);
750313  int inode_dir_empty (inode_t * inode);
750314  ssize_t inode_file_read (inode_t * inode, off_t offset,
750315                           void *buffer, size_t count,
750316                           int *eof);
750317  ssize_t inode_file_write (inode_t * inode,
750318                            off_t offset,
750319                            const void *buffer, size_t count);
750320  int inode_free (inode_t * inode);
750321  blkcnt_t inode_fzones_read (inode_t * inode,
750322                              zno_t zone_start,
750323                              void *buffer, blkcnt_t blkcnt);
750324  blkcnt_t inode_fzones_write (inode_t * inode,
750325                               zno_t zone_start,
750326                               void *buffer, blkcnt_t blkcnt);
750327  inode_t *inode_get (dev_t device, ino_t ino);
750328  inode_t *inode_pipe_make (void);
750329  ssize_t inode_pipe_read (inode_t * inode, void *buffer,
750330                           size_t count, int *eof);
750331  ssize_t inode_pipe_write (inode_t * inode,
750332                            const void *buffer, size_t count);
750333  void inode_print (void);
750334  int inode_put (inode_t * inode);
750335  inode_t *inode_reference (dev_t device, ino_t ino);
750336  int inode_save (inode_t * inode);
750337  inode_t *inode_stdio_dev_make (dev_t device, mode_t mode);
750338  int inode_truncate (inode_t * inode);
750339  zno_t inode_zone (inode_t * inode, zno_t fzone, int write);
750340  //-------------------------------------------------------
750341  file_t *file_pipe_make (void);
750342  file_t *file_reference (int fno);
750343  file_t *file_stdio_dev_make (dev_t device, mode_t mode,
750344                               int oflags);
750345  //-------------------------------------------------------
750346  dev_t path_device (pid_t pid, const char *path);
750347  int path_fix (char *path);
750348  int path_full (const char *path,
750349                 const char *path_cwd, char *full_path);
750350  inode_t *path_inode (pid_t pid, const char *path);
750351  inode_t *path_inode_link (pid_t pid, const char *path,
750352                            inode_t * inode, mode_t mode);
750353  //-------------------------------------------------------
750354  int fd_dup (pid_t pid, int fdn_old, int fdn_min);
750355  fd_t *fd_reference (pid_t pid, int *fdn);
750356  //-------------------------------------------------------
750357  //
750358  // void  sock_put (sock_t *s);
750359  //
750360  #define  sock_put(s)  (s->active=0)
750361
750362  sock_t *sock_reference (int skn);
750363  h_port_t sock_free_port (void);
750364  //-------------------------------------------------------
750365
750366  #endif
```

94.5.1 kernel/fs/fd_dup.c

Si veda la sezione 93.6.1.

```
760001  #include <kernel/proc.h>
760002  #include <kernel/fs.h>
760003  #include <errno.h>
760004  #include <fcntl.h>
760005  //-------------------------------------------------------
760006  int
760007  fd_dup (pid_t pid, int fdn_old, int fdn_min)
760008  {
760009    proc_t *ps;
760010    int fdn_new;
760011    //
760012    // Verify argument.
760013    //
760014    if (fdn_min < 0 || fdn_min >= OPEN_MAX)
760015      {
760016        errset (EINVAL);  // Invalid argument.
760017        return (-1);
760018      }
760019    //
760020    // Get process.
760021    //
760022    ps = proc_reference (pid);
760023    //
760024    // Verify if 'fdn_old' is a valid value.
760025    //
760026    if (fdn_old < 0 ||
760027        fdn_old >= OPEN_MAX || ps->fd[fdn_old].file == NULL)
760028      {
760029        errset (EBADF);   // Bad file descriptor.
760030        return (-1);
760031      }
760032    //
760033    // Find the first free slot and duplicate the file
760034    // descriptor.
760035    //
760036    for (fdn_new = fdn_min; fdn_new < OPEN_MAX; fdn_new++)
760037      {
760038        if (ps->fd[fdn_new].file == NULL)
760039          {
760040            ps->fd[fdn_new].fl_flags =
760041              ps->fd[fdn_old].fl_flags;
760042            ps->fd[fdn_new].fd_flags =
760043              ps->fd[fdn_old].fd_flags & ~FD_CLOEXEC;
760044            ps->fd[fdn_new].file = ps->fd[fdn_old].file;
760045            ps->fd[fdn_new].file->references++;
760046            return (fdn_new);
760047          }
760048      }
760049    //
760050    // No fd slot available.
760051    //
760052    errset (EMFILE);      // Too many open files.
760053    return (-1);
760054  }
```

94.5.2 kernel/fs/fd_reference.c

Si veda la sezione 93.6.2.

```
770001  #include <kernel/proc.h>
770002  #include <kernel/lib_k.h>
770003  #include <errno.h>
770004  //-------------------------------------------------------
770005  fd_t *
770006  fd_reference (pid_t pid, int *fdn)
770007  {
770008    proc_t *ps;
770009    //
770010    // Get process.
770011    //
770012    ps = proc_reference (pid);
770013    //
770014    // See what to do.
770015    //
770016    if (*fdn < 0)
770017      {
770018        //
770019        // Find the first free slot.
770020        //
770021        for (*fdn = 0; *fdn < OPEN_MAX; (*fdn)++)
770022          {
770023            if (ps->fd[*fdn].file == NULL)
770024              {
770025                return (&(ps->fd[*fdn]));
770026              }
770027          }
770028        *fdn = -1;
770029        return (NULL);
770030      }
770031    else
770032      {
770033        if (*fdn < OPEN_MAX)
770034          {
770035            //
770036            // Might return even a free file descriptor.
770037            //
770038            return (&(ps->fd[*fdn]));
770039          }
```

```
770040        else
770041          {
770042            return (NULL);
770043          }
770044      }
770045  }
```

94.5.3 kernel/fs/file_pipe_make.c

«

Si veda la sezione 93.6.4.

```
780001  #include <kernel/proc.h>
780002  #include <errno.h>
780003  #include <fcntl.h>
780004  //-----------------------------------------------------------
780005  file_t *
780006  file_pipe_make (void)
780007  {
780008    inode_t *inode;
780009    file_t *file;
780010    //
780011    // Try to allocate a device inode.
780012    //
780013    inode = inode_pipe_make ();
780014    if (inode == NULL)
780015      {
780016        //
780017        // Variable 'errno' is already set by
780018        // 'inode_stdio_dev_make()'.
780019        //
780020        errset (errno);
780021        return (NULL);
780022      }
780023    //
780024    // Inode allocated: need to allocate the system file
780025    // item.
780026    //
780027    file = file_reference (-1);
780028    if (file == NULL)
780029      {
780030        //
780031        // Remove the inode and return an error.
780032        //
780033        inode_put (inode);
780034        errset (ENFILE);  // Too many files open in
780035        // system.
780036        return (NULL);
780037      }
780038    //
780039    // Fill with data the system file item.
780040    //
780041    file->references = 2;
780042    file->oflags = (O_RDONLY | O_WRONLY);
780043    file->inode = inode;
780044    //
780045    // Return system file pointer.
780046    //
780047    return (file);
780048  }
```

94.5.4 kernel/fs/file_reference.c

«

Si veda la sezione 93.6.5.

```
790001  #include <kernel/proc.h>
790002  #include <errno.h>
790003  #include <fcntl.h>
790004  //-----------------------------------------------------------
790005  file_t *
790006  file_reference (int fno)
790007  {
790008    //
790009    // Check type of request.
790010    //
790011    if (fno < 0)
790012      {
790013        //
790014        // Find a free slot.
790015        //
790016        for (fno = 0; fno < FILE_MAX_SLOTS; fno++)
790017          {
790018            if (file_table[fno].references <= 0)
790019              {
790020                return (&file_table[fno]);
790021              }
790022          }
790023        return (NULL);
790024      }
790025    else if (fno > FILE_MAX_SLOTS)
790026      {
790027        return (NULL);
790028      }
790029    else
790030      {
790031        return (&file_table[fno]);
790032      }
790033  }
```


94.5.5 kernel/fs/file_stdio_dev_make.c

«

Si veda la sezione 93.6.6.

```
800001  #include <kernel/proc.h>
800002  #include <errno.h>
800003  #include <fcntl.h>
800004  //-----------------------------------------------------------
800005  file_t *
800006  file_stdio_dev_make (dev_t device, mode_t mode, int oflags)
800007  {
800008    inode_t *inode;
800009    file_t *file;
800010    //
800011    // Try to allocate a device inode.
800012    //
800013    inode = inode_stdio_dev_make (device, mode);
800014    if (inode == NULL)
800015      {
800016        //
800017        // Variable 'errno' is already set by
800018        // 'inode_stdio_dev_make()'.
800019        //
800020        errset (errno);
800021        return (NULL);
800022      }
800023    //
800024    // Inode allocated: need to allocate the system file
800025    // item.
800026    //
800027    file = file_reference (-1);
800028    if (file == NULL)
800029      {
800030        //
800031        // Remove the inode and return an error.
800032        //
800033        inode_put (inode);
800034        errset (ENFILE);  // Too many files open in
800035        // system.
800036        return (NULL);
800037      }
800038    //
800039    // Fill with data the system file item.
800040    //
800041    file->references = 1;
800042    file->oflags = (oflags & (O_RDONLY | O_WRONLY));
800043    file->inode = inode;
800044    //
800045    // Return system file pointer.
800046    //
800047    return (file);
800048  }
```

94.5.6 kernel/fs/fs_init.c

«

Si veda la sezione 93.6.3.

```
810001  #include <kernel/fs.h>
810002  #include <string.h>
810003  //-----------------------------------------------------------
810004  void
810005  fs_init (void)
810006  {
810007    int s;
810008    int i;
810009    int f;
810010    //
810011    for (s = 0; s < SB_MAX_SLOTS; s++)
810012      {
810013        sb_table[s].device = 0;
810014        sb_table[s].inode_mounted_on = NULL;
810015      }
810016    //
810017    for (i = 0; i < INODE_MAX_SLOTS; i++)
810018      {
810019        inode_table[i].references = 0;
```

```
810020      }
810021    //
810022    for (f = 0; f < FILE_MAX_SLOTS; f++)
810023      {
810024        file_table[f].references = 0;
810025        file_table[f].inode = NULL;
810026        file_table[f].sock = NULL;
810027      }
810028    //
810029    // Reset the socket table with 0x00.
810030    //
810031    memset (sock_table, 0x00, sizeof (sock_table));
810032  }
```

94.5.7 kernel/fs/fs_public.c

«

Si veda la sezione 93.6.

```
820001  #include <kernel/fs.h>
820002  //-----------------------------------------------------------
820003  sb_t sb_table[SB_MAX_SLOTS];
820004  file_t file_table[FILE_MAX_SLOTS];
820005  inode_t inode_table[INODE_MAX_SLOTS];
820006  sock_t sock_table[SOCK_MAX_SLOTS];
820007  //-----------------------------------------------------------
```

94.5.8 kernel/fs/inode_alloc.c

«

Si veda la sezione 93.6.7.

```
830001  #include <kernel/fs.h>
830002  #include <errno.h>
830003  #include <kernel/lib_k.h>
830004  #include <kernel/lib_s.h>
830005  //-----------------------------------------------------------
830006  inode_t *
830007  inode_alloc (dev_t device, mode_t mode, uid_t uid,
830008               gid_t gid)
830009  {
830010    sb_t *sb;
830011    inode_t *inode;
830012    int m;        // Index inside the inode map.
830013    int map_element;
830014    int map_bit;
830015    int map_mask;
830016    ino_t ino;
830017    //
830018    // Check for arguments.
830019    //
830020    if (mode == 0)
830021      {
830022        errset (EINVAL);  // Invalid argument.
830023        return (NULL);
830024      }
830025    //
830026    // Get the super block from the known device.
830027    //
830028    sb = sb_reference (device);
830029    if (sb == NULL)
830030      {
830031        errset (ENODEV);  // No such device.
830032        return (NULL);
830033      }
830034    //
830035    // Find a free inode.
830036    //
830037    while (1)
830038      {
830039        //
830040        // Scan the inode bit map, to find a free inode
830041        // for new allocation.
830042        //
830043        for (m = 0; m < (SB_MAP_INODE_SIZE * 16); m++)
830044          {
830045            map_element = m / 16;
830046            map_bit = m % 16;
830047            map_mask = 1 << map_bit;
830048            if (!(sb->map_inode[map_element] & map_mask))
830049              {
830050                //
830051                // Found a free element: change the map
830052                // to
830053                // allocate the inode.
830054                //
830055                sb->map_inode[map_element] |= map_mask;
830056                sb->changed = 1;
830057                ino = m;  // Found a free inode:
```



```
830058                break;    // exit the scan loop.
830059              }
830060          }
830061        //
830062        // Check if the scan was successful.
830063        //
830064        if (ino == 0)
830065          {
830066            errset (ENOSPC);      // No space left on
830067            // device.
830068            return (NULL);
830069          }
830070        //
830071        // The inode was allocated inside the map in
830072        // memory.
830073        //
830074        inode = inode_get (device, ino);
830075        if (inode == NULL)
830076          {
830077            errset (ENFILE);      // Too many files open
830078            // in system.
830079            return (NULL);
830080          }
830081        //
830082        // Verify if the inode is really free: if it
830083        // isn't, must save
830084        // it to disk.
830085        //
830086        if (inode->size > 0 || inode->links > 0)
830087          {
830088            //
830089            // Strange: should not have a size! Check if
830090            // there are even
830091            // links. Please note that 255 links (that
830092            // is -1) is to be
830093            // considered a free inode, marked in a
830094            // special way for some
830095            // unknown reason. Currently, 'LINK_MAX' is
830096            // equal to 254,
830097            // for that reason.
830098            //
830099            if (inode->links > 0 && inode->links < LINK_MAX)
830100              {
830101                //
830102                // Tell something.
830103                //
830104                k_printf ("kernel alert: device %04x: "
830105                          "found \"free\" inode %i "
830106                          "that still has size %i "
830107                          "and %i links!\n",
830108                          device, ino, inode->size,
830109                          inode->links);
830110                //
830111                // The inode must be set again to free,
830112                // inside
830113                // the bit map.
830114                //
830115                map_element = ino / 16;
830116                map_bit = ino % 16;
830117                map_mask = 1 << map_bit;
830118                sb->map_inode[map_element] &= ~map_mask;
830119                sb->changed = 1;
830120                //
830121                // Try to fix: reset all to zero.
830122                //
830123                inode->mode = 0;
830124                inode->uid = 0;
830125                inode->gid = 0;
830126                inode->time = 0;
830127                inode->links = 0;
830128                inode->size = 0;
830129                inode->direct[0] = 0;
830130                inode->direct[1] = 0;
830131                inode->direct[2] = 0;
830132                inode->direct[3] = 0;
830133                inode->direct[4] = 0;
830134                inode->direct[5] = 0;
830135                inode->direct[6] = 0;
830136                inode->indirect1 = 0;
830137                inode->indirect2 = 0;
830138                inode->changed = 1;
830139                //
830140                // Save fixed inode to disk.
830141                //
830142                inode_put (inode);
830143                continue;
830144              }
```

```
830145            else
830146              {
830147                //
830148                // Truncate the inode, save and break.
830149                //
830150                inode_truncate (inode);
830151                inode_save (inode);
830152                break;
830153              }
830154          }
830155        else
830156          {
830157            //
830158            // Considering free the inode found.
830159            //
830160            break;
830161          }
830162      }
830163    //
830164    // Put data inside the inode.
830165    //
830166    inode->mode = mode;
830167    inode->uid = uid;
830168    inode->gid = gid;
830169    inode->size = 0;
830170    inode->time = s_time ((pid_t) 0, NULL);
830171    inode->links = 0;
830172    inode->changed = 1;
830173    //
830174    // Save the inode.
830175    //
830176    inode_save (inode);
830177    //
830178    // Return the inode pointer.
830179    //
830180    return (inode);
830181  }
```

94.5.9 kernel/fs/inode_check.c

Si veda la sezione 93.6.8.

```
840001  #include <kernel/fs.h>
840002  #include <errno.h>
840003  #include <kernel/lib_k.h>
840004  //---------------------------------------------------------
840005  int
840006  inode_check (inode_t * inode, mode_t type, int perm,
840007               uid_t uid, gid_t gid)
840008  {
840009    //
840010    // Ensure that the variable 'type' has only the
840011    // requested file type.
840012    //
840013    type = (type & S_IFMT);
840014    //
840015    // Check inode argument.
840016    //
840017    if (inode == NULL)
840018      {
840019        errset (EINVAL);  // Invalid argument.
840020        return (-1);
840021      }
840022    //
840023    // The inode is not NULL: verify that the inode is
840024    // of a type
840025    // allowed (the parameter 'type' can hold more than
840026    // one
840027    // possibility).
840028    //
840029    if (!(inode->mode & type))
840030      {
840031        errset (E_FILE_TYPE);     // The file type is
840032        // not
840033        return (-1);      // the expected one.
840034      }
840035    //
840036    // The file type is correct.
840037    //
840038    if (inode->uid != 0 && uid == 0)
840039      {
840040        return (0);       // The root user has all
840041        // permissions.
840042      }
840043    //
840044    // The user is not root or the inode is owned by
840045    // root.
840046    //
840047    if (inode->uid == uid)
840048      {
840049        //
840050        // The user own the inode and must check user
840051        // permissions.
840052        //
840053        perm = (perm << 6);
840054        if ((inode->mode & perm) ^ perm)
840055          {
840056            errset (EACCES);      // Permission denied.
840057            return (-1);
840058          }
840059        else
840060          {
840061            return (0);
840062          }
840063      }
840064    //
840065    // The user does not own the inode: the group
840066    // permissions are
840067    // checked.
840068    //
840069    if (inode->gid == gid)
840070      {
840071        //
840072        // The group own the inode and must check user
840073        // permissions.
840074        //
840075        perm = (perm << 3);
840076        if ((inode->mode & perm) ^ perm)
840077          {
840078            errset (EACCES);      // Permission denied.
840079            return (-1);
840080          }
840081        else
840082          {
840083            return (0);
840084          }
840085      }
840086    //
840087    // The user and the group do not own the inode: the
840088    // other
840089    // permissions are checked.
840090    //
840091    if ((inode->mode & perm) ^ perm)
840092      {
840093        errset (EACCES);  // Permission denied.
840094        return (-1);
840095      }
840096    else
840097      {
840098        return (0);
840099      }
840100  }
```

94.5.10 kernel/fs/inode_dir_empty.c

Si veda la sezione 93.6.9.

```
850001  #include <kernel/fs.h>
850002  #include <errno.h>
850003  #include <kernel/lib_k.h>
850004  //---------------------------------------------------------
850005  int
850006  inode_dir_empty (inode_t * inode)
850007  {
850008    off_t start;
850009    char buffer[SB_MAX_ZONE_SIZE];
850010    directory_t *dir;
850011    ssize_t size_read;
850012    int d;        // Directory buffer index.
850013    //
850014    // Check argument: must be a directory.
850015    //
850016    if (inode == NULL || !S_ISDIR (inode->mode))
850017      {
850018        errset (EINVAL);  // Invalid argument.
850019        return (0);       // false
850020      }
850021    //
850022    // Read the directory content: if an item is present
850023    // (except '.' and
850024    // '..'), the directory is not empty.
850025    //
850026    for (start = 0;
850027         start < inode->size; start += inode->sb->blksize)
```

```
850028      {
850029        size_read =
850030          inode_file_read (inode, start, buffer,
850031                           inode->sb->blksize, NULL);
850032        if (size_read < sizeof (directory_t))
850033          {
850034            break;
850035          }
850036        //
850037        // Scan the directory portion just read.
850038        //
850039        dir = (directory_t *) buffer;
850040        //
850041        for (d = 0; d < size_read;
850042             d += (sizeof (directory_t)), dir++)
850043          {
850044            if (dir->ino != 0 &&
850045                strncmp (dir->name, ".", NAME_MAX) != 0
850046                && strncmp (dir->name, "..", NAME_MAX) != 0)
850047              {
850048                //
850049                // There is an item and the directory is
850050                // not empty.
850051                //
850052                return (0);       // false
850053              }
850054          }
850055      }
850056    //
850057    // Nothing was found; good!
850058    //
850059    return (1);   // true
850060  }
```

94.5.11 kernel/fs/inode_file_read.c

«

Si veda la sezione 93.6.10.

```
860001  #include <kernel/fs.h>
860002  #include <errno.h>
860003  #include <kernel/lib_k.h>
860004  //-----------------------------------------------------------
860005  ssize_t
860006  inode_file_read (inode_t * inode, off_t offset,
860007                   void *buffer, size_t count, int *eof)
860008  {
860009    unsigned char *destination = (unsigned char *) buffer;
860010    unsigned char zone_buffer[SB_MAX_ZONE_SIZE];
860011    blkcnt_t blkcnt_read;
860012    off_t off_fzone;      // File zone offset.
860013    off_t off_buffer;     // Destination buffer offset.
860014    ssize_t size_read;    // Byte transfer counter.
860015    zno_t fzone;
860016    off_t off_end;
860017    //
860018    // The inode pointer must be valid, and
860019    // the start byte must be positive.
860020    //
860021    if (inode == NULL || offset < 0)
860022      {
860023        errset (EINVAL);  // Invalid argument.
860024        return ((ssize_t) - 1);
860025      }
860026    //
860027    // Check if the start address is inside the file
860028    // size. This is not
860029    // an error, but zero bytes are read and '*eof' is
860030    // set. Otherwise,
860031    // '*eof' is reset.
860032    //
860033    if (offset >= inode->size)
860034      {
860035        (eof != NULL) ? *eof = 1 : 0;
860036        return (0);
860037      }
860038    else
860039      {
860040        (eof != NULL) ? *eof = 0 : 0;
860041      }
860042    //
860043    // Adjust, if necessary, the size of read, because
860044    // it cannot be
860045    // larger than the actual file size. The variable
860046    // 'off_end' is
860047    // used to calculate the position *after* the
860048    // requested read.
860049    // Remember that the first file position is byte
```



```
860050    // zero; so,
860051    // the byte index inside the file goes from zero to
860052    // inode->size -1.
860053    //
860054    off_end = offset;
860055    off_end += count;
860056    if (off_end > inode->size)
860057      {
860058        count = (inode->size - offset);
860059      }
860060    //
860061    // Read the first file-zone inside the zone buffer.
860062    //
860063    fzone = offset / inode->sb->blksize;
860064    off_fzone = offset % inode->sb->blksize;
860065    blkcnt_read =
860066      inode_fzones_read (inode, fzone, zone_buffer,
860067                         (blkcnt_t) 1);
860068    if (blkcnt_read <= 0)
860069      {
860070        //
860071        // Sorry!
860072        //
860073
860074        k_printf
860075          ("inode_fzones_read (inode, fzone %i,... )\n",
860076           fzone);
860077
860078        errset (EUNKNOWN);
860079        return (0);       // Zero bytes read!
860080      }
860081    //
860082    // The first file-zone was read: copy it inside the
860083    // destination
860084    // buffer and continue reading the other zones
860085    // needed. Variables
860086    // 'off_buffer' (destination buffer index) and
860087    // 'size_read' (copy
860088    // byte counter) must be reset here. Variable
860089    // 'off_fzone' is already
860090    // set with the initial offset inside 'zone_buffer'.
860091    //
860092    off_buffer = 0;
860093    size_read = 0;
860094    //
860095    while (count)
860096      {
860097        //
860098        // Copy the zone buffer into the destination.
860099        // Variables
860100        // 'off_fzone', 'off_buffer' and 'size_read'
860101        // must not be
860102        // initialized inside the loop.
860103        //
860104        for (;
860105             off_fzone < inode->sb->blksize && count > 0;
860106             off_fzone++, off_buffer++, size_read++,
860107             count--, offset++)
860108          {
860109            destination[off_buffer] = zone_buffer[off_fzone];
860110          }
860111        //
860112        // If not all the bytes are copied, read the
860113        // next file-zone.
860114        //
860115        if (count)
860116          {
860117            //
860118            // Read another file-zone inside the zone
860119            // buffer.
860120            // Again, the function 'inode_fzones_read()'
860121            // might
860122            // return a null pointer, but the variable
860123            // 'errno' tells if
860124            // it is really an error. For this reason,
860125            // the variable
860126            // 'errno' must be reset before the read,
860127            // and checked after
860128            // it.
860129            //
860130            fzone = offset / inode->sb->blksize;
860131            off_fzone = offset % inode->sb->blksize;
860132            blkcnt_read =
860133              inode_fzones_read (inode, fzone,
860134                                 zone_buffer, (blkcnt_t) 1);
860135            if (blkcnt_read <= 0)
860136              {
```

```
860137                //
860138                // Sorry: only 'size_read' bytes read!
860139                //
860140                errset (EUNKNOWN);
860141                return (size_read);
860142              }
860143          }
860144      }
860145    //
860146    // The requested size was read completely.
860147    //
860148    return (size_read);
860149  }
```

94.5.12 kernel/fs/inode_file_write.c

Si veda la sezione 93.6.11.

```
870001  #include <kernel/fs.h>
870002  #include <errno.h>
870003  #include <kernel/lib_k.h>
870004  //-----------------------------------------------------------
870005  ssize_t
870006  inode_file_write (inode_t * inode, off_t offset,
870007                    const void *buffer, size_t count)
870008  {
870009    unsigned char *buffer_source = (unsigned char *) buffer;
870010    unsigned char buffer_zone[SB_MAX_ZONE_SIZE];
870011    off_t off_fzone;      // File zone offset.
870012    off_t off_source;     // Source buffer offset.
870013    ssize_t size_copied;  // Byte transfer counter.
870014    ssize_t size_written; // Byte written counter.
870015    zno_t fzone;
870016    zno_t zone;
870017    blkcnt_t blkcnt_read;
870018    int status;
870019    //
870020    // The inode pointer must be valid, and
870021    // the start byte must be positive.
870022    //
870023    if (inode == NULL || offset < 0)
870024      {
870025        errset (EINVAL);  // Invalid argument.
870026        return ((ssize_t) - 1);
870027      }
870028    //
870029    // Read a zone, modify it with the source buffer,
870030    // then write it back
870031    // and continue reading and writing other zones if
870032    // needed.
870033    //
870034    for (size_written = 0, off_source = 0, size_copied =
870035         0; count > 0; size_written += size_copied)
870036      {
870037        //
870038        // Read the next file-zone inside the zone
870039        // buffer: the function
870040        // 'inode_zone()' is used to create
870041        // automatically the zone, if
870042        // it does not exist.
870043        //
870044        fzone = offset / inode->sb->blksize;
870045        off_fzone = offset % inode->sb->blksize;
870046        zone = inode_zone (inode, fzone, 1);
870047        if (zone == 0)
870048          {
870049            //
870050            // Return previously written bytes. The
870051            // variable 'errno' is
870052            // already set by 'inode_zone()'.
870053            //
870054            return (size_written);
870055          }
870056        blkcnt_read =
870057          inode_fzones_read (inode, fzone, buffer_zone,
870058                             (blkcnt_t) 1);
870059        if (blkcnt_read <= 0)
870060          {
870061            //
870062            // Even if the value is zero, there is a
870063            // problem reading the
870064            // zone to be overwritten (because
870065            // 'inode_zone()' should
870066            // have already created such zone). The
870067            // variable 'errno' is
870068            // already set by 'inode_fzones_read()'.
870069            //
```



```
870070            return ((ssize_t) - 1);
870071          }
870072        //
870073        // The zone was successfully loaded inside the
870074        // buffer: overwrite
870075        // the zone buffer with the source buffer.
870076        //
870077        for (size_copied = 0;
870078             off_fzone < inode->sb->blksize && count > 0;
870079             off_fzone++, off_source++, size_copied++,
870080             count--, offset++)
870081          {
870082            buffer_zone[off_fzone] =
870083              buffer_source[off_source];
870084          }
870085        //
870086        // Save the zone.
870087        //
870088        status = zone_write (inode->sb, zone, buffer_zone);
870089        if (status != 0)
870090          {
870091            //
870092            // Cannot save the zone: return the size
870093            // already written.
870094            // The variable 'errno' is already set by
870095            // 'zone_write()'.
870096            //
870097            return (size_written);
870098          }
870099        //
870100        // Zone saved: update the file size if necessary
870101        // (and the inode
870102        // too).
870103        //
870104        if (inode->size <= offset)
870105          {
870106            inode->size = offset;
870107            inode->changed = 1;
870108            inode_save (inode);
870109          }
870110      }
870111    //
870112    // All done successfully: return the value.
870113    //
870114    return (size_written);
870115  }
```

94.5.13 kernel/fs/inode_free.c

Si veda la sezione 93.6.12.

```
880001  #include <kernel/fs.h>
880002  #include <errno.h>
880003  #include <kernel/lib_k.h>
880004  //-----------------------------------------------------------
880005  int
880006  inode_free (inode_t * inode)
880007  {
880008    int map_element;
880009    int map_bit;
880010    int map_mask;
880011    //
880012    if (inode == NULL)
880013      {
880014        errset (EINVAL);  // Invalid argument.
880015        return (-1);
880016      }
880017    //
880018    map_element = inode->ino / 16;
880019    map_bit = inode->ino % 16;
880020    map_mask = 1 << map_bit;
880021    //
880022    if (inode->sb->map_inode[map_element] & map_mask)
880023      {
880024        inode->sb->map_inode[map_element] -= map_mask;
880025        inode->sb->changed = 1;
880026      }
880027    //
880028    inode->mode = 0;
880029    inode->uid = 0;
880030    inode->gid = 0;
880031    inode->size = 0;
880032    inode->time = 0;
880033    inode->links = 0;
880034    inode->changed = 1;
880035    inode->references = 0;
880036    //
```

```
880037    return (inode_save (inode));
880038  }
```

94.5.14 kernel/fs/inode_fzones_read.c

Si veda la sezione 93.6.13.

```
890001  #include <kernel/fs.h>
890002  #include <errno.h>
890003  #include <kernel/lib_k.h>
890004  //-------------------------------------------------------
890005  blkcnt_t
890006  inode_fzones_read (inode_t * inode, zno_t zone_start,
890007                     void *buffer, blkcnt_t blkcnt)
890008  {
890009    unsigned char *destination = (unsigned char *) buffer;
890010    int status;   // 'zone_read()' return value.
890011    blkcnt_t blkcnt_read; // Zone counter/index.
890012    zno_t zone;
890013    zno_t fzone;
890014    //
890015    // Read the zones into the destination buffer.
890016    //
890017    for (blkcnt_read = 0, fzone = zone_start;
890018         blkcnt_read < blkcnt; blkcnt_read++, fzone++)
890019      {
890020        //
890021        // Calculate the zone number, from the
890022        // file-zone, reading the
890023        // inode. If a zone is not really allocated, the
890024        // result is zero
890025        // and is valid.
890026        //
890027        zone = inode_zone (inode, fzone, 0);
890028        if (zone == ((zno_t) - 1))
890029          {
890030            //
890031            // This is an error. Return the read zones
890032            // quantity.
890033            //
890034            errset (EUNKNOWN);
890035            return (blkcnt_read);
890036          }
890037        //
890038        // Update the destination buffer pointer.
890039        //
890040        destination += (blkcnt_read * inode->sb->blksize);
890041        //
890042        // Read the zone inside the destination buffer,
890043        // but if the zone
890044        // is zero, a zeroed zone must be filled.
890045        //
890046        if (zone == 0)
890047          {
890048            memset (destination, 0,
890049                    (size_t) inode->sb->blksize);
890050          }
890051        else
890052          {
890053            status = zone_read (inode->sb, zone, destination);
890054            if (status != 0)
890055              {
890056                //
890057                // Could not read the requested zone:
890058                // return the zones
890059                // read correctly.
890060                //
890061                errset (EIO);     // I/O error.
890062                return (blkcnt_read);
890063              }
890064          }
890065      }
890066    //
890067    // All zones read correctly inside the buffer.
890068    //
890069    return (blkcnt_read);
890070  }
```

94.5.15 kernel/fs/inode_fzones_write.c

Si veda la sezione 93.6.13.

```
900001  #include <kernel/fs.h>
900002  #include <errno.h>
900003  #include <kernel/lib_k.h>
900004  //-------------------------------------------------------
900005  blkcnt_t
900006  inode_fzones_write (inode_t * inode, zno_t zone_start,
900007                      void *buffer, blkcnt_t blkcnt)
900008  {
900009    unsigned char *source = (unsigned char *) buffer;
900010    int status;   // 'zone_read()' return value.
900011    blkcnt_t blkcnt_written;      // Written zones
900012    // counter.
900013    zno_t zone;
900014    zno_t fzone;
900015    //
900016    // Write the zones into the destination buffer.
900017    //
900018    for (blkcnt_written = 0, fzone = zone_start;
900019         blkcnt_written < blkcnt; blkcnt_written++, fzone++)
900020      {
900021        //
900022        // Find real zone from file-zone.
900023        //
900024        zone = inode_zone (inode, fzone, 1);
900025        if (zone == 0 || zone == ((zno_t) - 1))
900026          {
900027            //
900028            // Function 'inode_zone()' should allocate
900029            // automatically
900030            // a missing zone and should return a valid
900031            // zone or
900032            // (zno_t) -1. Anyway, even if a zero zone
900033            // is returned,
900034            // it is an error. Return the
900035            // 'blkcnt_written' value.
900036            //
900037            return (blkcnt_written);
900038          }
900039        //
900040        // Update the source buffer pointer for the next
900041        // zone write.
900042        //
900043        source += (blkcnt_written * inode->sb->blksize);
900044        //
900045        // Write the zone from the buffer content.
900046        //
900047        status = zone_write (inode->sb, zone, source);
900048        if (status != 0)
900049          {
900050            //
900051            // Cannot write the zone. Return
900052            // 'size_written_zone' value.
900053            //
900054            return (blkcnt_written);
900055          }
900056      }
900057    //
900058    // All zones read correctly inside the buffer.
900059    //
900060    return (blkcnt_written);
900061  }
```

94.5.16 kernel/fs/inode_get.c

Si veda la sezione 93.6.15.

```
910001  #include <kernel/fs.h>
910002  #include <errno.h>
910003  #include <kernel/lib_k.h>
910004  #include <kernel/dev.h>
910005  //-------------------------------------------------------
910006  inode_t *
910007  inode_get (dev_t device, ino_t ino)
910008  {
910009    sb_t *sb;
910010    inode_t *inode;
910011    unsigned long int start;
910012    size_t size;
910013    ssize_t n;
910014    int status;
910015    //
910016    // Verify if the root file system inode was
910017    // requested.
910018    //
910019    if (device == 0 && ino == 1)
910020      {
910021        //
910022        // Get root file system inode.
910023        //
910024        inode = inode_reference (device, ino);
910025        if (inode == NULL)
910026          {
```

```
910027          //
910028          // The file system root directory inode is
910029          // not yet loaded:
910030          // get the first super block.
910031          //
910032          sb = sb_reference ((dev_t) 0);
910033          if (sb == NULL || sb->device == 0)
910034            {
910035              //
910036              // This error should never happen.
910037              //
910038              errset (EUNKNOWN);        // Unknown
910039              // error.
910040              return (NULL);
910041            }
910042          //
910043          // Load the file system root directory inode
910044          // (recursive
910045          // call).
910046          //
910047          inode = inode_get (sb->device, (ino_t) 1);
910048          if (inode == NULL)
910049            {
910050              //
910051              // This error should never happen.
910052              //
910053              errset (EUNKNOWN);        // Unknown
910054              // error.
910055              return (NULL);
910056            }
910057          //
910058          // Return the directory inode.
910059          //
910060          return (inode);
910061        }
910062      else
910063        {
910064          //
910065          // The file system root directory inode is
910066          // already
910067          // available.
910068          //
910069          if (inode->references >= INODE_MAX_REFERENCES)
910070            {
910071              errset (ENFILE);  // Too many files open
910072              // in system.
910073              return (NULL);
910074            }
910075          else
910076            {
910077              inode->references++;
910078              return (inode);
910079            }
910080        }
910081    }
910082  //
910083  // A common device-inode pair was requested: try to
910084  // find an already
910085  // cached inode.
910086  //
910087  inode = inode_reference (device, ino);
910088  if (inode != NULL)
910089    {
910090      if (inode->references >= INODE_MAX_REFERENCES)
910091        {
910092          errset (ENFILE);      // Too many files open
910093          // in system.
910094          return (NULL);
910095        }
910096      else
910097        {
910098          inode->references++;
910099          return (inode);
910100        }
910101    }
910102  //
910103  // The inode is not yet available: get super block.
910104  //
910105  sb = sb_reference (device);
910106  if (sb == NULL)
910107    {
910108      errset (ENODEV);  // No such device.
910109      return (NULL);
910110    }
910111  //
910112  // The super block is available, but the inode is
910113  // not yet cached.
```

```
910114  // Verify if the inode map reports it as allocated.
910115  //
910116  status = sb_inode_status (sb, ino);
910117  if (!status)
910118    {
910119      //
910120      // The inode is not allocated and cannot be
910121      // loaded.
910122      //
910123      errset (ENOENT);  // No such file or directory.
910124      return (NULL);
910125    }
910126  //
910127  // The inode was not already cached, but is
910128  // considered as allocated
910129  // inside the inode map. Find a free slot to load
910130  // the inode inside
910131  // the inode table (in memory).
910132  //
910133  inode = inode_reference ((dev_t) - 1, (ino_t) - 1);
910134  if (inode == NULL)
910135    {
910136      errset (ENFILE);  // Too many files open in
910137      // system.
910138      return (NULL);
910139    }
910140  //
910141  // A free inode slot was found. The inode must be
910142  // loaded.
910143  // Calculate the memory inode size, to be saved
910144  // inside the file
910145  // system: the administrative inode data, as it is
910146  // saved inside
910147  // the file system. The 'inode_t' type is bigger
910148  // than the real
910149  // inode administrative size, because it contains
910150  // more data, that is
910151  // not saved on disk.
910152  //
910153  size = offsetof (inode_t, sb);
910154  //
910155  // Calculating start position for read.
910156  //
910157  // [1] Boot block.
910158  // [2] Super block.
910159  // [3] Inode bit map.
910160  // [4] Zone bit map.
910161  // [5] Previous inodes: consider that the inode zero
910162  // is
910163  // present in the inode map, but not in the inode
910164  // table.
910165  //
910166  start = 1024; // [1]
910167  start += 1024;        // [2]
910168  start += (sb->map_inode_blocks * 1024);       // [3]
910169  start += (sb->map_zone_blocks * 1024);        // [4]
910170  start += ((ino - 1) * size);  // [5]
910171  //
910172  // Read inode from disk.
910173  //
910174  n =
910175    dev_io ((pid_t) - 1, device, DEV_READ, start,
910176            inode, size, NULL);
910177  if (n != size)
910178    {
910179      errset (EIO);     // I/O error.
910180      return (NULL);
910181    }
910182  //
910183  // The inode was read: add some data to the working
910184  // copy in memory.
910185  //
910186  inode->sb = sb;
910187  inode->sb_attached = NULL;
910188  inode->ino = ino;
910189  inode->references = 1;
910190  inode->changed = 0;
910191  //
910192  inode->blkcnt = inode->size;
910193  inode->blkcnt /= sb->blksize;
910194  if (inode->size % sb->blksize)
910195    {
910196      inode->blkcnt++;
910197    }
910198  //
910199  inode->pipe_dir = 1;  // Pipes must start with
910200  // write.
```

```
910201    inode->pipe_off_read = 0;
910202    inode->pipe_off_write = 0;
910203    inode->pipe_ref_read = 0;
910204    inode->pipe_ref_write = 0;
910205    //
910206    // Return the inode pointer.
910207    //
910208    return (inode);
910209  }
```

94.5.17 kernel/fs/inode_pipe_make.c

Si veda la sezione 93.6.16.

```
920001  #include <kernel/fs.h>
920002  #include <sys/stat.h>
920003  #include <errno.h>
920004  #include <kernel/lib_k.h>
920005  #include <kernel/lib_s.h>
920006  //-----------------------------------------------------------
920007  inode_t *
920008  inode_pipe_make (void)
920009  {
920010    inode_t *inode;
920011    //
920012    // Find a free inode.
920013    //
920014    inode = inode_reference ((dev_t) - 1, (ino_t) - 1);
920015    if (inode == NULL)
920016      {
920017        //
920018        // No free slot available.
920019        //
920020        errset (ENFILE);  // Too many files open in
920021        // system.
920022        return (NULL);
920023      }
920024    //
920025    // Put data inside the inode. Please note that
920026    // 'inode->ino' must be
920027    // zero, because it is necessary to recognize it as
920028    // an internal
920029    // inode with no file system. Otherwise, with a
920030    // value different than
920031    // zero, 'inode_put()' will try to remove it. [*]
920032    //
920033    inode->mode = S_IFIFO;
920034    inode->uid = 0;
920035    inode->gid = 0;
920036    inode->size = 0;
920037    inode->time = 0;
920038    inode->links = 0;
920039    inode->direct[0] = 0;
920040    inode->direct[1] = 0;
920041    inode->direct[2] = 0;
920042    inode->direct[3] = 0;
920043    inode->direct[4] = 0;
920044    inode->direct[5] = 0;
920045    inode->direct[6] = 0;
920046    inode->indirect1 = 0;
920047    inode->indirect2 = 0;
920048    inode->sb_attached = NULL;
920049    inode->sb = 0;
920050    inode->ino = 0;       // Must be zero. [*]
920051    inode->blkcnt = 0;
920052    inode->references = 1;
920053    inode->changed = 0;
920054    inode->pipe_dir = 1;  // Must start with write.
920055    inode->pipe_off_read = 0;
920056    inode->pipe_off_write = 0;
920057    inode->pipe_ref_read = 0;
920058    inode->pipe_ref_write = 0;
920059    //
920060    // Add all access permissions.
920061    //
920062    inode->mode |= (S_IRWXU | S_IRWXG | S_IRWXO);
920063    //
920064    // Return the inode pointer.
920065    //
920066    return (inode);
920067  }
```


94.5.18 kernel/fs/inode_pipe_read.c

Si veda la sezione 93.6.17.

```
930001  #include <kernel/fs.h>
930002  #include <errno.h>
930003  //-----------------------------------------------------------
930004  ssize_t
930005  inode_pipe_read (inode_t * inode, void *buffer,
930006                   size_t count, int *eof)
930007  {
930008    unsigned char *buffer_s;
930009    unsigned char *buffer_d = buffer;
930010    int i;
930011    //
930012    // The inode pointer must be valid.
930013    //
930014    if (inode == NULL)
930015      {
930016        errset (EINVAL);  // Invalid argument.
930017        return ((ssize_t) - 1);
930018      }
930019    //
930020    // Check the current pipe direction and see if can
930021    // be
930022    // read something.
930023    //
930024    if (inode->pipe_dir)
930025      {
930026        //
930027        // Write: if indexes are the same, cannot read
930028        // anything.
930029        //
930030        if (inode->pipe_off_write == inode->pipe_off_read)
930031          {
930032            //
930033            // Cannot read.
930034            //
930035            if (inode->pipe_ref_write == 0)
930036              {
930037                if (eof != NULL)
930038                  {
930039                    *eof = 1;
930040                  }
930041              }
930042            return ((ssize_t) 0);
930043          }
930044      }
930045    else
930046      {
930047        //
930048        // Read: the pipe is waiting for a read.
930049        //
930050        ;
930051      }
930052    //
930053    // Might read something. Set the pointer to the
930054    // source buffer,
930055    // that is the area used for direct zones, including
930056    // first
930057    // indirect pointers (total: (7+2)*2 = 18 bytes).
930058    //
930059    buffer_s = (void *) &(inode->direct[0]);
930060    //
930061    i = 0;
930062    //
930063    if (inode->pipe_off_read >= inode->pipe_off_write)
930064      {
930065        for (; i < count; i++)
930066          {
930067            if (inode->pipe_off_read < INODE_PIPE_BUFFER_SIZE)
930068              {
930069                buffer_d[i] = buffer_s[inode->pipe_off_read];
930070                inode->pipe_off_read++;
930071              }
930072            else
930073              {
930074                inode->pipe_off_read = 0;
930075                break;
930076              }
930077          }
930078      }
930079    //
930080    if (inode->pipe_off_read < inode->pipe_off_write)
930081      {
930082        for (; i < count; i++)
930083          {
930084            if (inode->pipe_off_read < inode->pipe_off_write)
930085              {
```

```
930086                buffer_d[i] = buffer_s[inode->pipe_off_read];
930087                inode->pipe_off_read++;
930088              }
930089            else
930090              {
930091                break;
930092              }
930093          }
930094      }
930095    //
930096    // At this point, it is time to set the direction to
930097    // write;
930098    // it doesn't matter if the direction is already set
930099    // so.
930100    //
930101    if (inode->pipe_off_read == inode->pipe_off_write)
930102      {
930103        inode->pipe_dir = 1;
930104      }
930105    //
930106    // Ok.
930107    //
930108    return ((ssize_t) i);
930109  }
```

94.5.19 kernel/fs/inode_pipe_write.c

«

Si veda la sezione 93.6.18.

```
940001  #include <kernel/fs.h>
940002  #include <errno.h>
940003  //-----------------------------------------------------------
940004  ssize_t
940005  inode_pipe_write (inode_t * inode, const void *buffer,
940006                    size_t count)
940007  {
940008    const unsigned char *buffer_s = buffer;
940009    unsigned char *buffer_d;
940010    int i;
940011    //
940012    // The inode pointer must be valid.
940013    //
940014    if (inode == NULL)
940015      {
940016        errset (EINVAL);  // Invalid argument.
940017        return ((ssize_t) - 1);
940018      }
940019    //
940020    // Check the current pipe direction and see if can
940021    // be
940022    // written something.
940023    //
940024    if (inode->pipe_dir)
940025      {
940026        //
940027        // Write: the pipe is waiting for a write.
940028        //
940029        ;
940030      }
940031    else
940032      {
940033        //
940034        // Read: if indexes are the same, cannot write
940035        // anything.
940036        //
940037        if (inode->pipe_off_write == inode->pipe_off_read)
940038          {
940039            //
940040            // Cannot write. More checks will be made by
940041            // 's_write()'.
940042            //
940043            return ((ssize_t) 0);
940044          }
940045      }
940046    //
940047    // Might write something. Set the pointer to the
940048    // destination buffer,
940049    // that is the area used for direct zones, including
940050    // first indirect
940051    // pointers (total: (7+2)*2 = 18 bytes).
940052    //
940053    buffer_d = (void *) &(inode->direct[0]);
940054    //
940055    i = 0;
940056    //
940057    if (inode->pipe_off_write >= inode->pipe_off_read)
940058      {
```



```
940059        for (; i < count; i++)
940060          {
940061            if (inode->pipe_off_write <
940062                INODE_PIPE_BUFFER_SIZE)
940063              {
940064                buffer_d[inode->pipe_off_write] = buffer_s[i];
940065                inode->pipe_off_write++;
940066              }
940067            else
940068              {
940069                inode->pipe_off_write = 0;
940070                break;
940071              }
940072          }
940073      }
940074    //
940075    if (inode->pipe_off_write < inode->pipe_off_read)
940076      {
940077        for (; i < count; i++)
940078          {
940079            if (inode->pipe_off_write < inode->pipe_off_read)
940080              {
940081                buffer_d[inode->pipe_off_write] = buffer_s[i];
940082                inode->pipe_off_write++;
940083              }
940084            else
940085              {
940086                break;
940087              }
940088          }
940089      }
940090    //
940091    // At this point, it is time to set the direction to
940092    // read;
940093    // it doesn't matter if the direction is already set
940094    // so.
940095    //
940096    if (inode->pipe_off_write == inode->pipe_off_read)
940097      {
940098        inode->pipe_dir = 0;
940099      }
940100    //
940101    // Ok.
940102    //
940103    return ((ssize_t) i);
940104  }
```

94.5.20 kernel/fs/inode_print.c

«

Si veda la sezione 93.6.19.

```
950001  #include <sys/os32.h>
950002  #include <kernel/fs.h>
950003  #include <kernel/lib_k.h>
950004  #include <time.h>
950005  //-----------------------------------------------------------
950006  void
950007  inode_print (void)
950008  {
950009    int i;
950010    dev_t device_attached = 0;
950011    time_t time;
950012    struct tm *timeptr;
950013    char type;
950014    dev_t device;
950015    //
950016    k_printf
950017      (" dev   ino ref c mntd t mode  uid gid size Kib "
950018       "date       time     lnk dirct[0]\n");
950019    //
950020    for (i = 0; i < INODE_MAX_SLOTS; i++)
950021      {
950022        if (inode_table[i].references <= 0)
950023          {
950024            continue;
950025          }
950026        //
950027        // Calculate modification time.
950028        //
950029        time = inode_table[i].time;
950030        //
950031        timeptr = gmtime (&time);
950032        //
950033        // Get type from mode.
950034        //
950035        if (S_ISBLK (inode_table[i].mode))
950036          type = 'b';
```

```
950037        else if (S_ISCHR (inode_table[i].mode))
950038          type = 'c';
950039        else if (S_ISFIFO (inode_table[i].mode))
950040          type = 'p';
950041        else if (S_ISREG (inode_table[i].mode))
950042          type = '-';
950043        else if (S_ISDIR (inode_table[i].mode))
950044          type = 'd';
950045        else if (S_ISLNK (inode_table[i].mode))
950046          type = 'l';
950047        else if (S_ISSOCK (inode_table[i].mode))
950048          type = 's';
950049        else
950050          type = '?';
950051        //
950052        // Is it a mount point?
950053        //
950054        if (inode_table[i].sb_attached != NULL)
950055          {
950056            device_attached =
950057              inode_table[i].sb_attached->device;
950058          }
950059        //
950060        // Is there a super block device?
950061        //
950062        if (inode_table[i].sb == NULL)
950063          {
950064            device = 0;
950065          }
950066        else
950067          {
950068            device = inode_table[i].sb->device;
950069          }
950070        //
950071        // Print data.
950072        //
950073        k_printf
950074          ("%04x %5i %3i %c %04x %c %04o %4i %3i %8i "
950075           "%4i.%02i.%02i %2i:%02i:%02i %3i %08x\n",
950076           (unsigned int) device,
950077           (unsigned int) inode_table[i].ino,
950078           (unsigned int) inode_table[i].references,
950079           (inode_table[i].changed ? '!' : ' '),
950080           (unsigned int) device_attached, type,
950081           (unsigned int) inode_table[i].mode,
950082           (unsigned int) inode_table[i].uid,
950083           (unsigned int) inode_table[i].gid,
950084           (unsigned int) (inode_table[i].size / 1024),
950085           timeptr->tm_year, timeptr->tm_mon,
950086           timeptr->tm_mday, timeptr->tm_hour,
950087           timeptr->tm_min, timeptr->tm_sec,
950088           (unsigned int) inode_table[i].links,
950089           (unsigned int) inode_table[i].direct[0]);
950090      }
950091  }
```

94.5.21 kernel/fs/inode_put.c

«

Si veda la sezione 93.6.20.

```
960001  #include <kernel/fs.h>
960002  #include <errno.h>
960003  #include <kernel/lib_k.h>
960004  //-----------------------------------------------------------
960005  int
960006  inode_put (inode_t * inode)
960007  {
960008    int status;
960009    //
960010    // Check for valid argument.
960011    //
960012    if (inode == NULL)
960013      {
960014        errset (EINVAL);  // Invalid argument.
960015        return (-1);
960016      }
960017    //
960018    // Check for valid references.
960019    //
960020    if (inode->references <= 0)
960021      {
960022        errset (EUNKNOWN);        // Cannot put an inode with
960023        return (-1);      // zero or negative
960024        // references.
960025      }
960026    //
960027    // Debug.
```



```
960028    //
960029    if (inode->ino != 0 && inode->sb->device == 0)
960030      {
960031        k_printf
960032          ("kernel alert: trying to close "
960033           "inode with device "
960034           "zero, but a number different than zero!\n");
960035        errset (EUNKNOWN);        // Cannot put an inode
960036        // with
960037        return (-1);      // zero or negative
960038        // references.
960039      }
960040    //
960041    // There is at least one reference: now the
960042    // references value is
960043    // reduced.
960044    //
960045    inode->references--;
960046    inode->changed = 1;
960047    //
960048    // If 'inode->ino' is zero, it means that the inode
960049    // was created in memory, but there is no file system
960050    // for it. For example, it might be a standard I/O
960051    // inode create automatically for a process.
960052    // Inodes with number zero cannot be removed from a
960053    // file system.
960054    //
960055    if (inode->ino == 0)
960056      {
960057        //
960058        // Nothing to do: just return.
960059        //
960060        return (0);
960061      }
960062    //
960063    // References counter might be zero.
960064    //
960065    if (inode->references == 0)
960066      {
960067        //
960068        // Check if the inode is to be deleted (until
960069        // there are
960070        // run time references, the inode cannot be
960071        // removed).
960072        //
960073        if (inode->links == 0
960074            || (S_ISDIR (inode->mode) && inode->links == 1))
960075          {
960076            //
960077            // The inode has no more run time references
960078            // and file system
960079            // links are also zero (or one for a
960080            // directory): remove it!
960081            //
960082            status = inode_truncate (inode);
960083            if (status != 0)
960084              {
960085                k_perror (NULL);
960086              }
960087            //
960088            inode_free (inode);
960089            return (0);
960090          }
960091      }
960092    //
960093    // Save inode to disk and return.
960094    //
960095    return (inode_save (inode));
960096  }
```

94.5.22 kernel/fs/inode_reference.c

«

Si veda la sezione 93.6.21.

```
970001  #include <kernel/fs.h>
970002  #include <errno.h>
970003  //-----------------------------------------------------------
970004  inode_t *
970005  inode_reference (dev_t device, ino_t ino)
970006  {
970007    int s;        // Slot index.
970008    sb_t *sb_table = sb_reference (0);
970009    //
970010    // If device is zero, and inode is zero, a reference
970011    // to the whole
970012    // table is returned.
970013    //
```

```
970014    if (device == 0 && ino == 0)
970015      {
970016        return (inode_table);
970017      }
970018    //
970019    // If device is ((dev_t) -1) and the inode is
970020    // ((ino_t) -1), a
970021    // reference to a free inode slot is returned.
970022    //
970023    if (device == (dev_t) - 1 && ino == ((ino_t) - 1))
970024      {
970025        for (s = 0; s < INODE_MAX_SLOTS; s++)
970026          {
970027            if (inode_table[s].references == 0)
970028              {
970029                return (&inode_table[s]);
970030              }
970031          }
970032        return (NULL);
970033      }
970034    //
970035    // If device is zero and the inode is 1, a reference
970036    // to the root
970037    // directory inode is returned.
970038    //
970039    if (device == 0 && ino == 1)
970040      {
970041        //
970042        // The super block table is to be scanned.
970043        //
970044        for (device = 0, s = 0; s < SB_MAX_SLOTS; s++)
970045          {
970046            if (sb_table[s].device != 0
970047                && (sb_table[s].inode_mounted_on->
970048                    sb_attached->device ==
970049                    sb_table[s].device))
970050              {
970051                device = sb_table[s].device;
970052                break;
970053              }
970054          }
970055        if (device == 0)
970056          {
970057            errset (E_CANNOT_FIND_ROOT_DEVICE);
970058            return (NULL);
970059          }
970060        //
970061        // Scan the inode table to find inode 1 and the
970062        // same device.
970063        //
970064        for (s = 0; s < INODE_MAX_SLOTS; s++)
970065          {
970066            if (inode_table[s].sb->device == device &&
970067                inode_table[s].ino == 1)
970068              {
970069                return (&inode_table[s]);
970070              }
970071          }
970072        //
970073        // Cannot find a root file system inode.
970074        //
970075        errset (E_CANNOT_FIND_ROOT_INODE);
970076        return (NULL);
970077      }
970078    //
970079    // A device and an inode number were selected: find
970080    // the inode
970081    // associated to it.
970082    //
970083    for (s = 0; s < INODE_MAX_SLOTS; s++)
970084      {
970085        if (inode_table[s].sb->device == device &&
970086            inode_table[s].ino == ino)
970087          {
970088            return (&inode_table[s]);
970089          }
970090      }
970091    //
970092    // The inode was not found.
970093    //
970094    return (NULL);
970095  }
```


94.5.23 kernel/fs/inode_save.c

Si veda la sezione 93.6.22.

```
980001  #include <kernel/fs.h>
980002  #include <errno.h>
980003  #include <kernel/dev.h>
980004  //-----------------------------------------------------------
980005  int
980006  inode_save (inode_t * inode)
980007  {
980008    size_t size;
980009    unsigned long int start;
980010    ssize_t n;
980011    //
980012    // Check for valid argument.
980013    //
980014    if (inode == NULL)
980015      {
980016        errset (EINVAL);  // Invalid argument.
980017        return (-1);
980018      }
980019    //
980020    // If the inode number is zero, no file system is
980021    // involved!
980022    //
980023    if (inode->ino == 0)
980024      {
980025        return (0);
980026      }
980027    //
980028    // Save the super block to disk.
980029    //
980030    sb_save (inode->sb);
980031    //
980032    // Save the inode to disk.
980033    //
980034    if (inode->changed)
980035      {
980036        size = offsetof (inode_t, sb);
980037        //
980038        // Calculating start position for write.
980039        //
980040        //
980041        // Boot block: 1024 bytes
980042        //
980043        start = 1024;
980044        //
980045        // Super block: + 1024 bytes
980046        //
980047        start += 1024;
980048        //
980049        // Inode bit map:
980050        //
980051        start += (inode->sb->map_inode_blocks * 1024);
980052        //
980053        // Zone bit map:
980054        //
980055        start += (inode->sb->map_zone_blocks * 1024);
980056        //
980057        // Previous inodes: consider that the inode zero
980058        // is present in the inode map, but not in the
980059        // inode table.
980060        //
980061        start += ((inode->ino - 1) * size);
980062        //
980063        // Write the inode.
980064        //
980065        n =
980066          dev_io ((pid_t) - 1, inode->sb->device,
980067                  DEV_WRITE, start, inode, size, NULL);
980068        //
980069        inode->changed = 0;
980070      }
980071    return (0);
980072  }
```

94.5.24 kernel/fs/inode_stdio_dev_make.c

Si veda la sezione 93.6.23.

```
990001  #include <kernel/fs.h>
990002  #include <errno.h>
990003  #include <kernel/lib_k.h>
990004  #include <kernel/lib_s.h>
990005  //-----------------------------------------------------------
990006  inode_t *
990007  inode_stdio_dev_make (dev_t device, mode_t mode)
```

```
990008  {
990009    inode_t *inode;
990010    //
990011    // Check for arguments.
990012    //
990013    if (mode == 0 || device == 0)
990014      {
990015        errset (EINVAL);  // Invalid argument.
990016        return (NULL);
990017      }
990018    //
990019    // Find a free inode.
990020    //
990021    inode = inode_reference ((dev_t) - 1, (ino_t) - 1);
990022    if (inode == NULL)
990023      {
990024        //
990025        // No free slot available.
990026        //
990027        errset (ENFILE);  // Too many files open in
990028        // system.
990029        return (NULL);
990030      }
990031    //
990032    // Put data inside the inode. Please note that
990033    // 'inode->ino' must be
990034    // zero, because it is necessary to recognize it as
990035    // an internal
990036    // inode with no file system. Otherwise, with a
990037    // value different than
990038    // zero, 'inode_put()' will try to remove it. [*]
990039    //
990040    inode->mode = mode;
990041    inode->uid = 0;
990042    inode->gid = 0;
990043    inode->size = 0;
990044    inode->time = s_time ((pid_t) 0, NULL);
990045    inode->links = 0;
990046    inode->direct[0] = device;
990047    inode->direct[1] = 0;
990048    inode->direct[2] = 0;
990049    inode->direct[3] = 0;
990050    inode->direct[4] = 0;
990051    inode->direct[5] = 0;
990052    inode->direct[6] = 0;
990053    inode->indirect1 = 0;
990054    inode->indirect2 = 0;
990055    inode->sb_attached = NULL;
990056    inode->sb = 0;
990057    inode->ino = 0;       // Must be zero. [*]
990058    inode->blkcnt = 0;
990059    inode->references = 1;
990060    inode->changed = 0;
990061    //
990062    // Add all access permissions.
990063    //
990064    inode->mode |= (S_IRWXU | S_IRWXG | S_IRWXO);
990065    //
990066    // Return the inode pointer.
990067    //
990068    return (inode);
990069  }
```

94.5.25 kernel/fs/inode_truncate.c

«

Si veda la sezione 93.6.24.

```
1000001  #include <kernel/fs.h>
1000002  #include <errno.h>
1000003  #include <kernel/lib_k.h>
1000004  //-----------------------------------------------------------
1000005  int
1000006  inode_truncate (inode_t * inode)
1000007  {
1000008    unsigned int indirect_zones;
1000009    zno_t zone_table1[INODE_MAX_INDIRECT_ZONES];
1000010    zno_t zone_table2[INODE_MAX_INDIRECT_ZONES];
1000011    unsigned int i;       // Direct index.
1000012    unsigned int i0;      // Single indirect index.
1000013    unsigned int i1;      // Double indirect first
1000014    // index.
1000015    unsigned int i2;      // Double indirect second
1000016    // index.
1000017    int status;   // 'zone_read()' return value.
1000018    //
1000019    // Check argument.
1000020    //
1000021    if (inode == NULL)
1000022      {
1000023        errset (EINVAL);
1000024        return (-1);
1000025      }
1000026    //
1000027    // Calculate how many indirect zone numbers are
1000028    // stored inside
1000029    // a zone: it depends on the zone size.
1000030    //
1000031    indirect_zones = inode->sb->blksize / 2;
1000032    //
1000033    // Scan and release direct zones. Errors are
1000034    // ignored.
1000035    //
1000036    for (i = 0; i < 7; i++)
1000037      {
1000038        zone_free (inode->sb, inode->direct[i]);
1000039        inode->direct[i] = 0;
1000040      }
1000041    //
1000042    // Scan single indirect zones, if present.
1000043    //
1000044    if (inode->blkcnt > 7 && inode->indirect1 != 0)
1000045      {
1000046        //
1000047        // There is a single indirect table to load.
1000048        // Errors are
1000049        // almost ignored.
1000050        //
1000051        status =
1000052          zone_read (inode->sb, inode->indirect1,
1000053                     zone_table1);
1000054        if (status == 0)
1000055          {
1000056            //
1000057            // Scan the table and remove zones.
1000058            //
1000059            for (i0 = 0; i0 < indirect_zones; i0++)
1000060              {
1000061                zone_free (inode->sb, zone_table1[i0]);
1000062              }
1000063          }
1000064        //
1000065        // Remove indirect table too.
1000066        //
1000067        zone_free (inode->sb, inode->indirect1);
1000068        //
1000069        // Clear single indirect reference inside the
1000070        // inode.
1000071        //
1000072        inode->indirect1 = 0;
1000073      }
1000074    //
1000075    // Scan double indirect zones, if present.
1000076    //
1000077    if (inode->blkcnt > (7 + indirect_zones)
1000078        && inode->indirect2 != 0)
1000079      {
1000080        //
1000081        // There is a double indirect table to load.
1000082        // Errors are
1000083        // almost ignored.
1000084        //
1000085        status =
1000086          zone_read (inode->sb, inode->indirect2,
1000087                     zone_table1);
1000088        if (status == 0)
1000089          {
1000090            //
1000091            // Scan the table and get second level
1000092            // indirection.
1000093            //
1000094            for (i1 = 0; i1 < indirect_zones; i1++)
1000095              {
1000096                if ((inode->blkcnt
1000097                     >
1000098                     (7 + indirect_zones +
1000099                      indirect_zones * i1))
1000100                    && zone_table1[i1] != 0)
1000101                  {
1000102                    //
1000103                    // There is a second level table to
1000104                    // load.
1000105                    //
1000106                    status =
1000107                      zone_read (inode->sb,
```

```
                                zone_table1[i1],
                                zone_table2);
                    if (status == 0)
                      {
                        //
                        // Release zones.
                        //
                        for (i2 = 0;
                             i2 < indirect_zones &&
                             (inode->blkcnt >
                              (7 + indirect_zones +
                               indirect_zones * i1 +
                               i2)); i2++)
                          {
                            zone_free (inode->sb,
                                       zone_table2[i2]);
                          }
                        //
                        // Remove second level indirect
                        // table.
                        //
                        zone_free (inode->sb,
                                   zone_table1[i1]);
                      }
                  }
              }
            //
            // Remove first level indirect table.
            //
            zone_free (inode->sb, inode->indirect2);
          }
        //
        // Clear single indirect reference inside the
        // inode.
        //
        inode->indirect2 = 0;
      }
    //
    // Update super block and inode data.
    //
    sb_save (inode->sb);
    inode->size = 0;
    inode->changed = 1;
    inode_save (inode);
    //
    // Successful return.
    //
    return (0);
}
```

94.5.26 kernel/fs/inode_zone.c

«

Si veda la sezione 93.6.25.

```
#include <kernel/fs.h>
#include <errno.h>
#include <kernel/lib_k.h>
//-----------------------------------------------------------
zno_t
inode_zone (inode_t * inode, zno_t fzone, int write)
{
  unsigned int indirect_zones;
  unsigned int allocated_zone;
  zno_t zone_table[INODE_MAX_INDIRECT_ZONES];
  char buffer[SB_MAX_ZONE_SIZE];
  unsigned int i0;      // Single indirect index.
  unsigned int i1;      // Double indirect first
  // index.
  unsigned int i2;      // Double indirect second
  // index.
  int status;
  zno_t zone_second;    // Second level table zone.
  //
  // Check to have a valid inode.
  //
  if (inode == NULL)
    {
      errset (EINVAL);
      return ((zno_t) - 1);
    }
  //
  // Calculate how many indirect zone numbers are
  // stored inside
  // a zone: it depends on the zone size.
  //
  indirect_zones = inode->sb->blksize / 2;
  //
  // Convert file-zone number into a zone number.
  //
  if (fzone < 7)
    {
      //
      // 0 <= fzone <= 6
      // The zone number is inside the direct zone
      // references.
      // Verify to have such zone.
      //
      if (inode->direct[fzone] == 0)
        {
          //
          // There is not such zone, but we do not
          // consider
          // it an error, because a file can be not
          // contiguous.
          //
          if (!write)
            {
              return ((zno_t) 0);
            }
          //
          // Must be allocated.
          //
          allocated_zone = zone_alloc (inode->sb);
          if (allocated_zone == 0)
            {
              //
              // Cannot allocate the zone. The
              // variable 'errno' is
              // set by 'zone_alloc()'.
              //
              return ((zno_t) - 1);
            }
          //
          // The zone is allocated: clear the zone and
          // save.
          //
          memset (buffer, 0, SB_MAX_ZONE_SIZE);
          status =
            zone_write (inode->sb, allocated_zone, buffer);
          if (status < 0)
            {
              //
              // Cannot overwrite the zone. The
              // variable 'errno' is
              // set by 'zone_write()'.
              //
              return ((zno_t) - 1);
            }
          //
          // The zone is allocated and cleared: save
          // the inode.
          //
          inode->direct[fzone] = allocated_zone;
          inode->changed = 1;
          status = inode_save (inode);
          if (status != 0)
            {
              //
              // Cannot save the inode. The variable
              // 'errno' is
              // set 'inode_save()'.
              //
              return ((zno_t) - 1);
            }
        }
      //
      // The zone is there: return it.
      //
      return (inode->direct[fzone]);
    }
  if (fzone < 7 + indirect_zones)
    {
      //
      // 7 <= fzone <= (6 + indirect_zones)
      // The zone number is inside the single indirect
      // zone
      // references: verify to have the indirect zone
      // table.
      //
      if (inode->indirect1 == 0)
        {
          //
          // There is not such zone, but it is not an
          // error.
```

```
1010121           //
1010122           if (!write)
1010123             {
1010124               return ((zno_t) 0);
1010125             }
1010126           //
1010127           // The first level of indirection must be
1010128           // initialized.
1010129           //
1010130           allocated_zone = zone_alloc (inode->sb);
1010131           if (allocated_zone == 0)
1010132             {
1010133               //
1010134               // Cannot allocate the zone for the
1010135               // indirection table:
1010136               // this is an error and the 'errno'
1010137               // value is produced
1010138               // by 'zone_alloc()'.
1010139               //
1010140               return ((zno_t) - 1);
1010141             }
1010142           //
1010143           // The zone for the indirection table is
1010144           // allocated:
1010145           // clear the zone and save.
1010146           //
1010147           memset (buffer, 0, SB_MAX_ZONE_SIZE);
1010148           status =
1010149             zone_write (inode->sb, allocated_zone, buffer);
1010150           if (status < 0)
1010151             {
1010152               //
1010153               // Cannot overwrite the zone. The
1010154               // variable 'errno' is
1010155               // set by 'zone_write()'.
1010156               //
1010157               return ((zno_t) - 1);
1010158             }
1010159           //
1010160           // The indirection table zone is allocated
1010161           // and cleared:
1010162           // save the inode.
1010163           //
1010164           inode->indirect1 = allocated_zone;
1010165           inode->changed = 1;
1010166           status = inode_save (inode);
1010167           if (status != 0)
1010168             {
1010169               //
1010170               // Cannot save the inode. This is an
1010171               // error and the value
1010172               // for 'errno' is produced by
1010173               // 'inode_save()'.
1010174               //
1010175               return ((zno_t) - 1);
1010176             }
1010177         }
1010178       //
1010179       // An indirect table is present inside the file
1010180       // system:
1010181       // load it.
1010182       //
1010183       status =
1010184         zone_read (inode->sb, inode->indirect1, zone_table);
1010185       if (status != 0)
1010186         {
1010187           //
1010188           // Cannot load the indirect table. This is
1010189           // an error and the
1010190           // value for 'errno' is assigned by function
1010191           // 'zone_read()'.
1010192           //
1010193           return ((zno_t) - 1);
1010194         }
1010195       //
1010196       // The indirect table was read. Calculate the
1010197       // index inside
1010198       // the table, for the requested zone.
1010199       //
1010200       i0 = (fzone - 7);
1010201       //
1010202       // Check if the zone is to be allocated.
1010203       //
1010204       if (zone_table[i0] == 0)
1010205         {
1010206           //
1010207           // There is not such zone, but it is not an
```

```
1010208           // error.
1010209           //
1010210           if (!write)
1010211             {
1010212               return ((zno_t) 0);
1010213             }
1010214           //
1010215           // The zone must be allocated.
1010216           //
1010217           allocated_zone = zone_alloc (inode->sb);
1010218           if (allocated_zone == 0)
1010219             {
1010220               //
1010221               // There is no space for the zone
1010222               // allocation. The
1010223               // variable 'errno' is already updated
1010224               // by
1010225               // 'zone_alloc()'.
1010226               //
1010227               return ((zno_t) - 1);
1010228             }
1010229           //
1010230           // The zone is allocated: clear the zone and
1010231           // save.
1010232           //
1010233           memset (buffer, 0, SB_MAX_ZONE_SIZE);
1010234           status =
1010235             zone_write (inode->sb, allocated_zone, buffer);
1010236           if (status < 0)
1010237             {
1010238               //
1010239               // Cannot overwrite the zone. The
1010240               // variable 'errno' is
1010241               // set by 'zone_write()'.
1010242               //
1010243               return ((zno_t) - 1);
1010244             }
1010245           //
1010246           // The zone is allocated and cleared: update
1010247           // the indirect
1010248           // zone table an save it. The inode is not
1010249           // modified,
1010250           // because the indirect table is outside.
1010251           //
1010252           zone_table[i0] = allocated_zone;
1010253           status =
1010254             zone_write (inode->sb, inode->indirect1,
1010255                         zone_table);
1010256           if (status != 0)
1010257             {
1010258               //
1010259               // Cannot save the zone. The variable
1010260               // 'errno' is already
1010261               // set by 'zone_write()'.
1010262               //
1010263               return ((zno_t) - 1);
1010264             }
1010265         }
1010266       //
1010267       // The zone is allocated.
1010268       //
1010269       return (zone_table[i0]);
1010270     }
1010271   else
1010272     {
1010273       //
1010274       // (7 + indirect_zones) <= fzone
1010275       // The zone number is inside the double indirect
1010276       // zone
1010277       // references.
1010278       // Verify to have the first level of second
1010279       // indirection.
1010280       //
1010281       if (inode->indirect2 == 0)
1010282         {
1010283           //
1010284           // There is not such zone, but it is not an
1010285           // error.
1010286           //
1010287           if (!write)
1010288             {
1010289               return ((zno_t) 0);
1010290             }
1010291           //
1010292           // The first level of second indirection
1010293           // must be
1010294           // initialized.
```

```
1010295           //
1010296           allocated_zone = zone_alloc (inode->sb);
1010297           if (allocated_zone == 0)
1010298             {
1010299               //
1010300               // Cannot allocate the zone. The
1010301               // variable 'errno' is
1010302               // set by 'zone_alloc()'.
1010303               //
1010304               return ((zno_t) - 1);
1010305             }
1010306           //
1010307           // The zone for the indirection table is
1010308           // allocated:
1010309           // clear the zone and save.
1010310           //
1010311           memset (buffer, 0, SB_MAX_ZONE_SIZE);
1010312           status =
1010313             zone_write (inode->sb, allocated_zone, buffer);
1010314           if (status < 0)
1010315             {
1010316               //
1010317               // Cannot overwrite the zone. The
1010318               // variable 'errno' is
1010319               // set by 'zone_write()'.
1010320               //
1010321               return ((zno_t) - 1);
1010322             }
1010323           //
1010324           // The zone for the indirection table is
1010325           // allocated and
1010326           // cleared: save the inode.
1010327           //
1010328           inode->indirect2 = allocated_zone;
1010329           inode->changed = 1;
1010330           status = inode_save (inode);
1010331           if (status != 0)
1010332             {
1010333               //
1010334               // Cannot save the inode. The variable
1010335               // 'errno' is
1010336               // set by 'inode_save()'.
1010337               //
1010338               return ((zno_t) - 1);
1010339             }
1010340         }
1010341       //
1010342       // The first level of second indirection is
1010343       // present:
1010344       // Read the second indirect table.
1010345       //
1010346       status =
1010347         zone_read (inode->sb, inode->indirect2, zone_table);
1010348       if (status != 0)
1010349         {
1010350           //
1010351           // Cannot read the second indirect table.
1010352           // The variable
1010353           // 'errno' is set by 'zone_read()'.
1010354           //
1010355           return ((zno_t) - 1);
1010356         }
1010357       //
1010358       // The first double indirect table was read:
1010359       // calculate
1010360       // indexes inside first and second level of
1010361       // table.
1010362       //
1010363       fzone -= 7;
1010364       fzone -= indirect_zones;
1010365       i1 = fzone / indirect_zones;
1010366       i2 = fzone % indirect_zones;
1010367       //
1010368       // Verify to have a second level.
1010369       //
1010370       if (zone_table[i1] == 0)
1010371         {
1010372           //
1010373           // There is not such zone, but it is not an
1010374           // error.
1010375           //
1010376           if (!write)
1010377             {
1010378               return ((zno_t) 0);
1010379             }
1010380           //
1010381           // The second level must be initialized.
```



```
1010382           //
1010383           allocated_zone = zone_alloc (inode->sb);
1010384           if (allocated_zone == 0)
1010385             {
1010386               //
1010387               // Cannot allocate the zone. The
1010388               // variable 'errno' is set
1010389               // by 'zone_alloc()'.
1010390               //
1010391               return ((zno_t) - 1);
1010392             }
1010393           //
1010394           // The zone for the indirection table is
1010395           // allocated:
1010396           // clear the zone and save.
1010397           //
1010398           memset (buffer, 0, SB_MAX_ZONE_SIZE);
1010399           status =
1010400             zone_write (inode->sb, allocated_zone, buffer);
1010401           if (status < 0)
1010402             {
1010403               //
1010404               // Cannot overwrite the zone. The
1010405               // variable 'errno' is
1010406               // set by 'zone_write()'.
1010407               //
1010408               return ((zno_t) - 1);
1010409             }
1010410           //
1010411           // Update the first level index and save it.
1010412           //
1010413           zone_table[i1] = allocated_zone;
1010414           status =
1010415             zone_write (inode->sb, inode->indirect2,
1010416                         zone_table);
1010417           if (status != 0)
1010418             {
1010419               //
1010420               // Cannot write the zone. The variable
1010421               // 'errno' is set
1010422               // by 'zone_write()'.
1010423               //
1010424               return ((zno_t) - 1);
1010425             }
1010426         }
1010427       //
1010428       // The second level can be read, overwriting the
1010429       // array
1010430       // 'zone_table[]'. The zone number for the
1010431       // second level
1010432       // indirection table is saved inside
1010433       // 'zone_second', before
1010434       // overwriting the array.
1010435       //
1010436       zone_second = zone_table[i1];
1010437       status =
1010438         zone_read (inode->sb, zone_second, zone_table);
1010439       if (status != 0)
1010440         {
1010441           //
1010442           // Cannot read the second level indirect
1010443           // table. The variable
1010444           // 'errno' is set by 'zone_read()'.
1010445           //
1010446           return ((zno_t) - 1);
1010447         }
1010448       //
1010449       // The second level was read and 'zone_table[]'
1010450       // is now
1010451       // such second one: check if the zone is to be
1010452       // allocated.
1010453       //
1010454       if (zone_table[i2] == 0)
1010455         {
1010456           //
1010457           // There is not such zone, but it is not an
1010458           // error.
1010459           //
1010460           if (!write)
1010461             {
1010462               return ((zno_t) 0);
1010463             }
1010464           //
1010465           // Must be allocated.
1010466           //
1010467           allocated_zone = zone_alloc (inode->sb);
1010468           if (allocated_zone == 0)
```

```
1010469             {
1010470               //
1010471               // Cannot allocate the zone. The
1010472               // variable 'errno' is set
1010473               // by 'zone_alloc()'.
1010474               //
1010475               return ((zno_t) - 1);
1010476             }
1010477           //
1010478           // The zone is allocated: clear the zone and
1010479           // save.
1010480           //
1010481           memset (buffer, 0, SB_MAX_ZONE_SIZE);
1010482           status =
1010483             zone_write (inode->sb, allocated_zone, buffer);
1010484           if (status < 0)
1010485             {
1010486               //
1010487               // Cannot overwrite the zone. The
1010488               // variable 'errno' is
1010489               // set by 'zone_write()'.
1010490               //
1010491               return ((zno_t) - 1);
1010492             }
1010493           //
1010494           // The zone was allocated and cleared:
1010495           // update the indirect
1010496           // zone table an save it. The inode is not
1010497           // modified, because
1010498           // the indirect table is outside.
1010499           //
1010500           zone_table[i2] = allocated_zone;
1010501           status =
1010502             zone_write (inode->sb, zone_second, zone_table);
1010503           if (status != 0)
1010504             {
1010505               //
1010506               // Cannot write the zone. The variable
1010507               // 'errno' is set
1010508               // by 'zone_write()'.
1010509               //
1010510               return ((zno_t) - 1);
1010511             }
1010512         }
1010513       //
1010514       // The zone is there: return the zone number.
1010515       //
1010516       return (zone_table[i2]);
1010517     }
1010518 }
```

94.5.27 kernel/fs/path_device.c

Si veda la sezione 93.6.38.

```
1020001 #include <kernel/fs.h>
1020002 #include <errno.h>
1020003 #include <kernel/proc.h>
1020004 //-------------------------------------------------------
1020005 dev_t
1020006 path_device (pid_t pid, const char *path)
1020007 {
1020008   proc_t *ps;
1020009   inode_t *inode;
1020010   dev_t device;
1020011   //
1020012   // Get process.
1020013   //
1020014   ps = proc_reference (pid);
1020015   //
1020016   inode = path_inode (pid, path);
1020017   if (inode == NULL)
1020018     {
1020019       errset (errno);
1020020       return ((dev_t) - 1);
1020021     }
1020022   //
1020023   if (!(S_ISBLK (inode->mode) || S_ISCHR (inode->mode)))
1020024     {
1020025       errset (ENODEV);  // No such device.
1020026       inode_put (inode);
1020027       return ((dev_t) - 1);
1020028     }
1020029   //
1020030   device = inode->direct[0];
1020031   inode_put (inode);
1020032   return (device);
1020033 }
```


94.5.28 kernel/fs/path_fix.c

Si veda la sezione 93.6.39.

```
1030001 #include <kernel/fs.h>
1030002 #include <errno.h>
1030003 #include <kernel/proc.h>
1030004 //-------------------------------------------------------
1030005 int
1030006 path_fix (char *path)
1030007 {
1030008   char new_path[PATH_MAX];
1030009   char *token[PATH_MAX / 4];
1030010   int t;        // Token index.
1030011   int token_size;       // Token array effective size.
1030012   int comp;     // String compare return value.
1030013   size_t path_size;     // Path string size.
1030014   //
1030015   // Initialize token search.
1030016   //
1030017   token[0] = strtok (path, "/");
1030018   //
1030019   // Scan tokens.
1030020   //
1030021   for (t = 0;
1030022        t < PATH_MAX / 4 && token[t] != NULL;
1030023        t++, token[t] = strtok (NULL, "/"))
1030024     {
1030025       //
1030026       // If current token is '.', just ignore it.
1030027       //
1030028       comp = strcmp (token[t], ".");
1030029       if (comp == 0)
1030030         {
1030031           t--;
1030032         }
1030033       //
1030034       // If current token is '..', remove previous
1030035       // token,
1030036       // if there is one.
1030037       //
1030038       comp = strcmp (token[t], "..");
1030039       if (comp == 0)
1030040         {
1030041           if (t > 0)
1030042             {
1030043               t -= 2;
1030044             }
1030045           else
1030046             {
1030047               t = -1;
1030048             }
1030049         }
1030050       //
1030051       // 't' will be incremented and another token
1030052       // will be
1030053       // found.
1030054       //
1030055     }
1030056   //
1030057   // Save the token array effective size.
1030058   //
1030059   token_size = t;
1030060   //
1030061   // Initialize the new path string.
1030062   //
1030063   new_path[0] = '\0';
1030064   //
1030065   // Build the new path string.
1030066   //
1030067   if (token_size > 0)
1030068     {
1030069       for (t = 0; t < token_size; t++)
1030070         {
1030071           path_size = strlen (new_path);
1030072           strncat (new_path, "/", 2);
1030073           strncat (new_path, token[t],
1030074                    PATH_MAX - path_size - 1);
1030075         }
1030076     }
1030077   else
1030078     {
1030079       strncat (new_path, "/", 2);
1030080     }
1030081   //
```

```
1030082   // Copy the new path into the original string.
1030083   //
1030084   strncpy (path, new_path, PATH_MAX);
1030085   //
1030086   // Return.
1030087   //
1030088   return (0);
1030089 }
```

94.5.29 kernel/fs/path_full.c

«

Si veda la sezione 93.6.40.

```
1040001 #include <kernel/fs.h>
1040002 #include <errno.h>
1040003 #include <kernel/proc.h>
1040004 //-------------------------------------------------------
1040005 int
1040006 path_full (const char *path, const char *path_cwd,
1040007           char *full_path)
1040008 {
1040009   unsigned int path_size;
1040010   //
1040011   // Check some arguments.
1040012   //
1040013   if (path == NULL || strlen (path) == 0
1040014       || full_path == NULL)
1040015     {
1040016       errset (EINVAL);  // Invalid argument.
1040017       return (-1);
1040018     }
1040019   //
1040020   // The main path and the receiving one are right.
1040021   // Now arrange to get a full path name.
1040022   //
1040023   if (path[0] == '/')
1040024     {
1040025       strncpy (full_path, path, PATH_MAX);
1040026       full_path[PATH_MAX - 1] = 0;
1040027     }
1040028   else
1040029     {
1040030       if (path_cwd == NULL || strlen (path_cwd) == 0)
1040031         {
1040032           errset (EINVAL);      // Invalid argument.
1040033           return (-1);
1040034         }
1040035       strncpy (full_path, path_cwd, PATH_MAX);
1040036       path_size = strlen (full_path);
1040037       strncat (full_path, "/", (PATH_MAX - path_size));
1040038       path_size = strlen (full_path);
1040039       strncat (full_path, path, (PATH_MAX - path_size));
1040040     }
1040041   //
1040042   // Fix path name so that it has no '..', '.', and no
1040043   // multiple '/'.
1040044   //
1040045   path_fix (full_path);
1040046   //
1040047   // Return.
1040048   //
1040049   return (0);
1040050 }
```

94.5.30 kernel/fs/path_inode.c

«

Si veda la sezione 93.6.41.

```
1050001 #include <kernel/fs.h>
1050002 #include <errno.h>
1050003 #include <kernel/proc.h>
1050004 #include <kernel/lib_k.h>
1050005 //-------------------------------------------------------
1050006 #define DIRECTORY_BUFFER_SIZE (SB_MAX_ZONE_SIZE/16)
1050007 //-------------------------------------------------------
1050008 inode_t *
1050009 path_inode (pid_t pid, const char *path)
1050010 {
1050011   proc_t *ps;
1050012   inode_t *inode;
1050013   dev_t device;
1050014   char full_path[PATH_MAX];
1050015   char *name;
1050016   char *next;
1050017   directory_t dir[DIRECTORY_BUFFER_SIZE];
1050018   char dir_name[NAME_MAX + 1];
1050019   off_t offset_dir;
```



```
1050020   ssize_t size_read;
1050021   size_t dir_size_read;
1050022   ssize_t size_to_read;
1050023   int comp;
1050024   int d;        // Directory index;
1050025   int status;   // inode_check() return status.
1050026   //
1050027   // Get process.
1050028   //
1050029   ps = proc_reference (pid);
1050030   //
1050031   // Arrange to get a packed full path name.
1050032   //
1050033   path_full (path, ps->path_cwd, full_path);
1050034   //
1050035   // Get the root file system inode.
1050036   //
1050037   inode = inode_get ((dev_t) 0, 1);
1050038   if (inode == NULL)
1050039     {
1050040       errset (errno);
1050041       return (NULL);
1050042     }
1050043   //
1050044   // Save the device number.
1050045   //
1050046   device = inode->sb->device;
1050047   //
1050048   // Variable 'inode' already points to the root file
1050049   // system inode:
1050050   // It must be a directory!
1050051   //
1050052   status =
1050053     inode_check (inode, S_IFDIR, 1, ps->euid, ps->egid);
1050054   if (status != 0)
1050055     {
1050056       //
1050057       // Variable 'errno' should be set by
1050058       // inode_check().
1050059       //
1050060       errset (errno);
1050061       inode_put (inode);
1050062       return (NULL);
1050063     }
1050064   //
1050065   // Initialize string scan: find the first path
1050066   // token, after the
1050067   // first '/'.
1050068   //
1050069   name = strtok (full_path, "/");
1050070   //
1050071   // If the original full path is just '/' the
1050072   // variable 'name'
1050073   // appears as a null pointer, and the variable
1050074   // 'inode' is already
1050075   // what we are looking for.
1050076   //
1050077   if (name == NULL)
1050078     {
1050079       return (inode);
1050080     }
1050081   //
1050082   // There is at least a name after '/' inside the
1050083   // original full
1050084   // path. A scan is going to start: the original
1050085   // value for variable
1050086   // 'inode' is a pointer to the root directory inode.
1050087   //
1050088   for (;;)
1050089     {
1050090       //
1050091       // Find next token.
1050092       //
1050093       next = strtok (NULL, "/");
1050094       //
1050095       // Read the directory from the current inode.
1050096       //
1050097       for (offset_dir = 0;; offset_dir += size_read)
1050098         {
1050099           size_to_read = DIRECTORY_BUFFER_SIZE;
1050100           //
1050101           if ((offset_dir + size_to_read) > inode->size)
1050102             {
1050103               size_to_read = inode->size - offset_dir;
1050104             }
1050105           //
1050106           size_read =
```

```
1050107             inode_file_read (inode, offset_dir, dir,
1050108                              size_to_read, NULL);
1050109           //
1050110           // The size read must be a multiple of 16.
1050111           //
1050112           size_read = ((size_read / 16) * 16);
1050113           //
1050114           // Check anyway if it is zero.
1050115           //
1050116           if (size_read == 0)
1050117             {
1050118               //
1050119               // The directory is ended: release the
1050120               // inode and return.
1050121               //
1050122               inode_put (inode);
1050123               errset (ENOENT);  // No such file or
1050124               // directory.
1050125               return (NULL);
1050126             }
1050127           //
1050128           // Calculate how many directory items we
1050129           // have read.
1050130           //
1050131           dir_size_read = size_read / 16;
1050132           //
1050133           // Scan the directory to find the current
1050134           // name.
1050135           //
1050136           for (d = 0; d < dir_size_read; d++)
1050137             {
1050138               //
1050139               // Ensure to have a null terminated
1050140               // string for
1050141               // the name found.
1050142               //
1050143               memcpy (dir_name, dir[d].name,
1050144                       (size_t) NAME_MAX);
1050145               dir_name[NAME_MAX] = 0;
1050146               //
1050147               comp = strcmp (name, dir_name);
1050148               if (comp == 0 && dir[d].ino != 0)
1050149                 {
1050150                   //
1050151                   // Found the name and verified that
1050152                   // it has a link to
1050153                   // a inode. Now release the
1050154                   // directory inode.
1050155                   //
1050156                   inode_put (inode);
1050157                   //
1050158                   // Get next inode and break the
1050159                   // loop.
1050160                   //
1050161                   inode = inode_get (device, dir[d].ino);
1050162                   break;
1050163                 }
1050164             }
1050165           //
1050166           // If index 'd' is in a valid range, the
1050167           // name was found.
1050168           //
1050169           if (d < dir_size_read)
1050170             {
1050171               //
1050172               // The name was found.
1050173               //
1050174               break;
1050175             }
1050176         }
1050177       //
1050178       // If the function is still working, a file or a
1050179       // directory
1050180       // was found: see if there is another name after
1050181       // this one
1050182       // to look for. If there isn't, just break the
1050183       // loop.
1050184       //
1050185       if (next == NULL)
1050186         {
1050187           //
1050188           // As no other tokens are to be found, break
1050189           // the loop.
1050190           //
1050191           break;
1050192         }
1050193       //
```



```
1050194       // As there is another name after the current
1050195       // one,
1050196       // the current file must be a directory.
1050197       //
1050198       status =
1050199         inode_check (inode, S_IFDIR, 1, ps->euid, ps->egid);
1050200       if (status != 0)
1050201         {
1050202           //
1050203           // Variable 'errno' is set by
1050204           // 'inode_check()'.
1050205           //
1050206           errset (errno);
1050207           inode_put (inode);
1050208           return (NULL);
1050209         }
1050210       //
1050211       // The inode is a directory and the user has the
1050212       // necessary
1050213       // permissions: check if it is a mount point and
1050214       // go to the
1050215       // new device root directory if necessary.
1050216       //
1050217       if (inode->sb_attached != NULL)
1050218         {
1050219           //
1050220           // Must find the root directory for the new
1050221           // device, and
1050222           // then go to that inode.
1050223           //
1050224           device = inode->sb_attached->device;
1050225           inode_put (inode);
1050226           inode = inode_get (device, 1);
1050227           status = inode_check (inode, S_IFDIR, 1,
1050228                                 ps->euid, ps->egid);
1050229           if (status != 0)
1050230             {
1050231               inode_put (inode);
1050232               return (NULL);
1050233             }
1050234         }
1050235       //
1050236       // As a directory was found, and another token
1050237       // follows it,
1050238       // must continue the token scan.
1050239       //
1050240       name = next;
1050241     }
1050242   //
1050243   // Current inode found is the file represented by
1050244   // the requested
1050245   // path.
1050246   //
1050247   return (inode);
1050248 }
```

94.5.31 kernel/fs/path_inode_link.c

Si veda la sezione 93.6.42.

```
1060001 #include <kernel/fs.h>
1060002 #include <errno.h>
1060003 #include <kernel/proc.h>
1060004 #include <libgen.h>
1060005 //-----------------------------------------------------------
1060006 inode_t *
1060007 path_inode_link (pid_t pid, const char *path,
1060008                  inode_t * inode, mode_t mode)
1060009 {
1060010   proc_t *ps;
1060011   char buffer[SB_MAX_ZONE_SIZE];
1060012   off_t start;
1060013   int d;        // Directory index.
1060014   ssize_t size_read;
1060015   ssize_t size_written;
1060016   directory_t *dir = (directory_t *) buffer;
1060017   char path_copy1[PATH_MAX];
1060018   char path_copy2[PATH_MAX];
1060019   char *path_directory;
1060020   char *path_name;
1060021   inode_t *inode_directory;
1060022   inode_t *inode_new;
1060023   dev_t device;
1060024   int status;
1060025   //
1060026   // Check arguments.
1060027   //
```

```
1060028   if (path == NULL || strlen (path) == 0)
1060029     {
1060030       errset (EINVAL);  // Invalid argument:
1060031       return (NULL);    // the path is mandatory.
1060032     }
1060033   //
1060034   if (inode == NULL && mode == 0)
1060035     {
1060036       errset (EINVAL);  // Invalid argument: if the
1060037       // inode is to
1060038       return (NULL);    // be created, the mode is
1060039       // mandatory.
1060040     }
1060041   //
1060042   if (inode != NULL)
1060043     {
1060044       if (mode != 0)
1060045         {
1060046           errset (EINVAL);      // Invalid argument:
1060047           // if the inode is
1060048           return (NULL);        // already present,
1060049           // the creation mode
1060050         }       // must not be given.
1060051       if (S_ISDIR (inode->mode))
1060052         {
1060053           errset (EPERM);       // Operation not
1060054           // permitted.
1060055           return (NULL);        // Refuse to link
1060056           // directory.
1060057         }
1060058       if (inode->links >= LINK_MAX)
1060059         {
1060060           errset (EMLINK);      // Too many links.
1060061           return (NULL);
1060062         }
1060063     }
1060064   //
1060065   // Get process.
1060066   //
1060067   ps = proc_reference (pid);
1060068   //
1060069   // If the destination path already exists, the link
1060070   // cannot be made.
1060071   // It does not matter if the inode is known or not.
1060072   //
1060073   inode_new = path_inode ((uid_t) 0, path);
1060074   if (inode_new != NULL)
1060075     {
1060076       //
1060077       // A file already exists with the same name.
1060078       //
1060079       inode_put (inode_new);
1060080       errset (EEXIST);  // File exists.
1060081       return (NULL);
1060082     }
1060083   //
1060084   // At this point, 'inode_new' is 'NULL'.
1060085   // Copy the source path inside the directory path
1060086   // and name arrays.
1060087   //
1060088   strncpy (path_copy1, path, PATH_MAX);
1060089   strncpy (path_copy2, path, PATH_MAX);
1060090   //
1060091   // Reduce to directory name and find the last name.
1060092   //
1060093   path_directory = dirname (path_copy1);
1060094   path_name = basename (path_copy2);
1060095   if (strlen (path_directory) == 0
1060096       || strlen (path_name) == 0)
1060097     {
1060098       errset (EACCES);  // Permission denied: maybe
1060099       // the
1060100       // original path is the root directory
1060101       // and cannot find a previous directory.
1060102       return (NULL);
1060103     }
1060104   //
1060105   // Get the directory inode.
1060106   //
1060107   inode_directory = path_inode (pid, path_directory);
1060108   if (inode_directory == NULL)
1060109     {
1060110       errset (errno);
1060111       return (NULL);
1060112     }
1060113   //
1060114   // Check if something is mounted on it.
```



```
1060115   //
1060116   if (inode_directory->sb_attached != NULL)
1060117     {
1060118       //
1060119       // Must select the right directory.
1060120       //
1060121       device = inode_directory->sb_attached->device;
1060122       inode_put (inode_directory);
1060123       inode_directory = inode_get (device, 1);
1060124       if (inode_directory == NULL)
1060125         {
1060126           return (NULL);
1060127         }
1060128     }
1060129   //
1060130   // If the inode to link is known, check if the
1060131   // selected directory
1060132   // has the same super block than the inode to link.
1060133   //
1060134   if (inode != NULL && inode_directory->sb != inode->sb)
1060135     {
1060136       inode_put (inode_directory);
1060137       errset (ENOENT);  // No such file or directory.
1060138       return (NULL);
1060139     }
1060140   //
1060141   // Check if write is allowed for the file system.
1060142   //
1060143   if (inode_directory->sb->options & MOUNT_RO)
1060144     {
1060145       inode_put (inode_directory);
1060146       errset (EROFS);   // Read-only file system.
1060147       return (NULL);
1060148     }
1060149   //
1060150   // Verify access permissions for the directory. The
1060151   // number "3" means
1060152   // that the user must have access permission and
1060153   // write permission:
1060154   // "-wx" == 2+1 == 3.
1060155   //
1060156   status = inode_check (inode_directory, S_IFDIR, 3,
1060157                         ps->euid, ps->egid);
1060158   if (status != 0)
1060159     {
1060160       inode_put (inode_directory);
1060161       return (NULL);
1060162     }
1060163   //
1060164   // If the inode to link was not specified, it must
1060165   // be created.
1060166   // From now on, the inode is referenced with the
1060167   // variable
1060168   // 'inode_new'.
1060169   //
1060170   inode_new = inode;
1060171   //
1060172   if (inode_new == NULL)
1060173     {
1060174       inode_new =
1060175         inode_alloc (inode_directory->sb->device, mode,
1060176                      ps->euid, ps->egid);
1060177       if (inode_new == NULL)
1060178         {
1060179           //
1060180           // The inode allocation failed, so, also the
1060181           // directory
1060182           // must be released, before return.
1060183           //
1060184           inode_put (inode_directory);
1060185           return (NULL);
1060186         }
1060187     }
1060188   //
1060189   // Read the directory content and try to add the new
1060190   // item.
1060191   //
1060192   for (start = 0;
1060193        start < inode_directory->size;
1060194        start += inode_directory->sb->blksize)
1060195     {
1060196       size_read =
1060197         inode_file_read (inode_directory, start,
1060198                          buffer,
1060199                          inode_directory->sb->blksize,
1060200                          NULL);
1060201       if (size_read < sizeof (directory_t))
```

```
1060202         {
1060203           break;
1060204         }
1060205       //
1060206       // Scan the directory portion just read, for an
1060207       // unused item.
1060208       //
1060209       dir = (directory_t *) buffer;
1060210       for (d = 0; d < size_read;
1060211            d += (sizeof (directory_t)), dir++)
1060212         {
1060213           if (dir->ino == 0)
1060214             {
1060215               //
1060216               // Found an empty directory item: link
1060217               // the inode.
1060218               //
1060219               dir->ino = inode_new->ino;
1060220               strncpy (dir->name, path_name, NAME_MAX);
1060221               inode_new->links++;
1060222               inode_new->changed = 1;
1060223               //
1060224               // Update the directory inside the file
1060225               // system.
1060226               //
1060227               size_written =
1060228                 inode_file_write (inode_directory,
1060229                                   start, buffer, size_read);
1060230               if (size_written != size_read)
1060231                 {
1060232                   //
1060233                   // Write problem: release the
1060234                   // directory and return.
1060235                   //
1060236                   inode_put (inode_directory);
1060237                   errset (EUNKNOWN);
1060238                   return (NULL);
1060239                 }
1060240               //
1060241               // Save the new inode, release the
1060242               // directory and return
1060243               // the linked inode.
1060244               //
1060245               inode_save (inode_new);
1060246               inode_put (inode_directory);
1060247               return (inode_new);
1060248             }
1060249         }
1060250     }
1060251   //
1060252   // The directory don't have a free item and one must
1060253   // be appended.
1060254   //
1060255   dir = (directory_t *) buffer;
1060256   start = inode_directory->size;
1060257   //
1060258   // Prepare the buffer with the link.
1060259   //
1060260   dir->ino = inode_new->ino;
1060261   strncpy (dir->name, path_name, NAME_MAX);
1060262   inode_new->links++;
1060263   inode_new->changed = 1;
1060264   //
1060265   // Append the buffer to the directory.
1060266   //
1060267   size_written =
1060268     inode_file_write (inode_directory, start, buffer,
1060269                       (sizeof (directory_t)));
1060270   if (size_written != (sizeof (directory_t)))
1060271     {
1060272       //
1060273       // Problem updating the directory: release it
1060274       // and return.
1060275       //
1060276       inode_put (inode_directory);
1060277       errset (EUNKNOWN);
1060278       return (NULL);
1060279     }
1060280   //
1060281   // Close access to the directory inode and save the
1060282   // other inode,
1060283   // with updated link count.
1060284   //
1060285   inode_put (inode_directory);
1060286   inode_save (inode_new);
1060287   //
1060288   // Return successfully.
```



```
1060289   //
1060290   return (inode_new);
1060291 }
```

94.5.32 kernel/fs/sb_inode_status.c

«

Si veda la sezione 93.6.26.

```
1070001 #include <kernel/fs.h>
1070002 #include <errno.h>
1070003 //-------------------------------------------------------
1070004 int
1070005 sb_inode_status (sb_t * sb, ino_t ino)
1070006 {
1070007   int map_element;
1070008   int map_bit;
1070009   int map_mask;
1070010   //
1070011   // Check arguments.
1070012   //
1070013   if (ino == 0 || sb == NULL)
1070014     {
1070015       errset (EINVAL);  // Invalid argument.
1070016       return (-1);
1070017     }
1070018   //
1070019   // Calculate the map element, the map bit and the
1070020   // map mask.
1070021   //
1070022   map_element = ino / 16;
1070023   map_bit = ino % 16;
1070024   map_mask = 1 << map_bit;
1070025   //
1070026   // Check the inode and return.
1070027   //
1070028   if (sb->map_inode[map_element] & map_mask)
1070029     {
1070030       return (1);       // True.
1070031     }
1070032   else
1070033     {
1070034       return (0);       // False.
1070035     }
1070036 }
```

94.5.33 kernel/fs/sb_mount.c

«

Si veda la sezione 93.6.27.

```
1080001 #include <kernel/fs.h>
1080002 #include <errno.h>
1080003 #include <kernel/dev.h>
1080004 #include <kernel/lib_k.h>
1080005 #include <kernel/dm.h>
1080006 #include <kernel/part.h>
1080007 //-------------------------------------------------------
1080008 sb_t *
1080009 sb_mount (dev_t device, inode_t ** inode_mnt, int options)
1080010 {
1080011   sb_t *sb;
1080012   ssize_t size_read;
1080013   addr_t start;
1080014   int m;
1080015   size_t size_sb;
1080016   size_t size_map;
1080017   int dev_major = major (device);
1080018   int dev_minor = minor (device);
1080019   int p = dev_minor & 0x000F;
1080020   int d = ((dev_minor & 0x00F0) >> 4);
1080021   //
1080022   // Find if it is already mounted.
1080023   //
1080024   sb = sb_reference (device);
1080025   if (sb != NULL)
1080026     {
1080027       errset (EBUSY);   // Device or resource busy:
1080028       // device
1080029       return (NULL);    // already mounted.
1080030     }
1080031   //
1080032   // Find if '*inode_mnt' is already mounting
1080033   // something.
1080034   //
1080035   if (*inode_mnt != NULL
1080036       && (*inode_mnt)->sb_attached != NULL)
1080037     {
1080038       errset (EBUSY);   // Device or resource busy:
```

```
1080039       // mount point
1080040       return (NULL);    // already used.
1080041     }
1080042   //
1080043   // If it is a partition, find if it can be mounted.
1080044   //
1080045   if ((dev_major == DEV_DM_MAJOR) && (p));
1080046   {
1080047     //
1080048     // It is a partition.
1080049     //
1080050     if (dm_table[d].part[p].type != PART_TYPE_MINIX)
1080051       {
1080052         errset (E_PART_TYPE_NOT_MINIX); // Not Minix!
1080053         return (NULL);  // Cannot mount.
1080054       }
1080055   }
1080056   //
1080057   // The inode is not yet mounting anything, or it is
1080058   // new: find a free
1080059   // slot inside the super block table.
1080060   //
1080061   sb = sb_reference ((dev_t) - 1);
1080062   if (sb == NULL)
1080063     {
1080064       errset (EBUSY);   // Device or resource busy:
1080065       return (NULL);    // no free slots.
1080066     }
1080067   //
1080068   // A free slot was found: the super block header
1080069   // must be loaded, but
1080070   // before it is necessary to calculate the header
1080071   // size to be read.
1080072   //
1080073   size_sb = offsetof (sb_t, device);
1080074   //
1080075   // Then fix the starting point.
1080076   //
1080077   start = 1024; // After boot block.
1080078   //
1080079   // Read the file system super block header.
1080080   //
1080081   size_read =
1080082     dev_io ((pid_t) 0, device, DEV_READ, start, sb,
1080083             size_sb, NULL);
1080084   if (size_read != size_sb)
1080085     {
1080086       errset (EIO);     // I/O error.
1080087       return (NULL);
1080088     }
1080089   //
1080090   // Save some more data.
1080091   //
1080092   sb->device = device;
1080093   sb->options = options;
1080094   sb->inode_mounted_on = *inode_mnt;
1080095   sb->blksize = (1024 << sb->log2_size_zone);
1080096   //
1080097   // Check if the super block data is valid.
1080098   //
1080099   if (sb->magic_number != 0x137F)
1080100     {
1080101       errset (ENODEV);  // No such device: unsupported
1080102       sb->device = 0;   // file system type.
1080103       return (NULL);
1080104     }
1080105   if (sb->map_inode_blocks > SB_MAX_INODE_BLOCKS)
1080106     {
1080107       errset (E_MAP_INODE_TOO_BIG);
1080108       return (NULL);
1080109     }
1080110   if (sb->map_zone_blocks > SB_MAX_ZONE_BLOCKS)
1080111     {
1080112       errset (E_MAP_ZONE_TOO_BIG);
1080113       return (NULL);
1080114     }
1080115   if (sb->blksize > SB_MAX_ZONE_SIZE)
1080116     {
1080117       errset (E_DATA_ZONE_TOO_BIG);
1080118       return (NULL);
1080119     }
1080120   //
1080121   // A right super block header was loaded from disk,
1080122   // now load the super block inode bit map.
1080123   //
1080124   start = 1024; // After boot block.
1080125   start += 1024;        // After super block.
```

```
1080126   //
1080127   // Reset map in memory before loading.
1080128   //
1080129   for (m = 0; m < SB_MAP_INODE_SIZE; m++)       // [2]
1080130     {
1080131       sb->map_inode[m] = 0xFFFF;        // [2]
1080132     }
1080133   size_map = sb->map_inode_blocks * 1024;
1080134   size_read =
1080135     dev_io ((pid_t) - 1, sb->device, DEV_READ, start,
1080136             sb->map_inode, size_map, NULL);
1080137   if (size_read != size_map)
1080138     {
1080139       errset (EIO);     // I/O error.
1080140       return (NULL);
1080141     }
1080142   //
1080143   // Load the super block zone bit map.
1080144   //
1080145   // After boot block:
1080146   //
1080147   start = 1024;
1080148   //
1080149   // After the super block:
1080150   //
1080151   start += 1024;
1080152   //
1080153   // After inode bit map:
1080154   //
1080155   start += (sb->map_inode_blocks * 1024);
1080156   //
1080157   // Reset map in memory, before loading.
1080158   //
1080159   for (m = 0; m < SB_MAP_ZONE_SIZE; m++)
1080160     {
1080161       sb->map_zone[m] = 0xFFFF;
1080162     }
1080163   //
1080164   size_map = sb->map_zone_blocks * 1024;
1080165   size_read =
1080166     dev_io ((pid_t) - 1, sb->device, DEV_READ, start,
1080167             sb->map_zone, size_map, NULL);
1080168   if (size_read != size_map)
1080169     {
1080170       errset (EIO);     // I/O error.
1080171       return (NULL);
1080172     }
1080173   //
1080174   // Check the inode that should mount the super
1080175   // block. If '*inode_mnt' is 'NULL', then it is meant
1080176   // to be the first mount of the root file system.
1080177   // In such case, the inode must be loaded too,
1080178   // and the value for '*inode_mnt' must be modified.
1080179   //
1080180   if (*inode_mnt == NULL)
1080181     {
1080182       *inode_mnt = inode_get (device, 1);
1080183     }
1080184   //
1080185   // Check for a valid value.
1080186   //
1080187   if (*inode_mnt == NULL)
1080188     {
1080189       //
1080190       // This is bad!
1080191       //
1080192       errset (EUNKNOWN);        // Unknown error.
1080193       return (NULL);
1080194     }
1080195   //
1080196   // A valid inode is available for the mount.
1080197   //
1080198   (*inode_mnt)->sb_attached = sb;
1080199   //
1080200   // Update the super block too.
1080201   //
1080202   sb->inode_mounted_on = *inode_mnt;
1080203   //
1080204   // Return the super block pointer.
1080205   //
1080206   return (sb);
1080207 }
```

94.5.34 kernel/fs/sb_print.c

Si veda la sezione 93.6.28.

```
#include <sys/os32.h>
#include <kernel/fs.h>
#include <kernel/lib_k.h>
//-----------------------------------------------------------
void
sb_print (void)
{
  int s;
  //
  k_printf
    ("     mnt               1st zone max file "
     " inode zone  \n");
  k_printf
    ("dev  dev  inodes blocks dz  size size KiB "
     "blocks blocks\n");
  //
  for (s = 0; s < SB_MAX_SLOTS; s++)
    {
      if (sb_table[s].device == 0)
        {
          continue;
        }
      k_printf
        ("%04x %04x % 6i % 6i % 3i "
         "% 4i % 8i % 6i % 6i\n",
         sb_table[s].device,
         sb_table[s].inode_mounted_on->sb_attached->device,
         sb_table[s].inodes, sb_table[s].zones,
         sb_table[s].first_data_zone,
         (1024 << sb_table[s].log2_size_zone),
         sb_table[s].max_file_size / 1024,
         sb_table[s].map_inode_blocks,
         sb_table[s].map_zone_blocks);
    }
}
```

94.5.35 kernel/fs/sb_reference.c

Si veda la sezione 93.6.29.

```
#include <kernel/fs.h>
#include <errno.h>
//-----------------------------------------------------------
sb_t *
sb_reference (dev_t device)
{
  int s;        // Slot index.
  //
  // If device is zero, a reference to the whole table
  // is returned.
  //
  if (device == 0)
    {
      return (sb_table);
    }
  //
  // If device is ((dev_t) -1), a reference to a free
  // slot is
  // returned.
  //
  if (device == ((dev_t) - 1))
    {
      for (s = 0; s < SB_MAX_SLOTS; s++)
        {
          if (sb_table[s].device == 0)
            {
              return (&sb_table[s]);
            }
        }
      return (NULL);
    }
  //
  // A device was selected: find the super block
  // associated to it.
  //
  for (s = 0; s < SB_MAX_SLOTS; s++)
    {
      if (sb_table[s].device == device)
        {
          return (&sb_table[s]);
        }
    }
  //
  // The super block was not found.
  //
  return (NULL);
}
```


94.5.36 kernel/fs/sb_save.c

Si veda la sezione 93.6.30.

```
#include <kernel/fs.h>
#include <errno.h>
#include <kernel/dev.h>
//-----------------------------------------------------------
int
sb_save (sb_t * sb)
{
  ssize_t size_written;
  addr_t start;
  size_t size_map;
  //
  // Check for valid argument.
  //
  if (sb == NULL)
    {
      errset (EINVAL);  // Invalid argument.
      return (-1);
    }
  //
  // Check if the super block changed for some reason
  // (only the inode and the zone maps can change
  // really).
  //
  if (!sb->changed)
    {
      //
      // Nothing to save.
      //
      return (0);
    }
  //
  // Something inside the super block changed: start
  // the procedure to save the inode map (recall that
  // the super block header is not saved, because it
  // never changes).
  //
  start = 1024; // After boot block.
  start += 1024;        // After super block.
  size_map = sb->map_inode_blocks * 1024;
  size_written =
    dev_io ((pid_t) - 1, sb->device, DEV_WRITE, start,
            sb->map_inode, size_map, NULL);
  if (size_written != size_map)
    {
      //
      // Error writing the map.
      //
      errset (EIO);     // I/O error.
      return (-1);
    }
  //
  // Start the procedure to save the zone map.
  //
  start = 1024; // After boot block.
  start += 1024;        // After super block.
  start += (sb->map_inode_blocks * 1024);       // After
  // inode bit
  // map.
  size_map = sb->map_zone_blocks * 1024;
  size_written =
    dev_io ((pid_t) - 1, sb->device, DEV_WRITE, start,
            sb->map_zone, size_map, NULL);
  if (size_written != size_map)
    {
      //
      // Error writing the map.
      //
      errset (EIO);     // I/O error.
      return (-1);
    }
  //
  // Super block saved.
  //
  sb->changed = 0;
  //
  return (0);
}
```

94.5.37 kernel/fs/sb_zone_status.c

Si veda la sezione 93.6.26.

```
#include <kernel/fs.h>
#include <errno.h>
//---------------------------------------------------------
int
sb_zone_status (sb_t * sb, zno_t zone)
{
  int map_element;
  int map_bit;
  int map_mask;
  //
  // Check arguments.
  //
  if (zone == 0 || sb == NULL)
    {
      errset (EINVAL);  // Invalid argument.
      return (-1);
    }
  //
  // Calculate the map element, the map bit and the
  // map mask.
  //
  map_element = zone / 16;
  map_bit = zone % 16;
  map_mask = 1 << map_bit;
  //
  // Check the zone and return.
  //
  if (sb->map_zone[map_element] & map_mask)
    {
      return (1);       // True.
    }
  else
    {
      return (0);       // False.
    }
}
```

94.5.38 kernel/fs/sock_free_port.c

Si veda la sezione 93.6.32.

```
#include <kernel/fs.h>
#include <errno.h>
#include <fcntl.h>
//---------------------------------------------------------
h_port_t
sock_free_port (void)
{
  int skn;
  h_port_t lport;
  //
  for (lport = 0xFFFF; lport >= 1024; lport--)
    {
      for (skn = 0; skn < SOCK_MAX_SLOTS; skn++)
        {
          if (sock_table[skn].lport == lport)
            {
              //
              // The port is used.
              //
              break;
            }
        }
      if (sock_table[skn].lport != lport)
        {
          //
          // The port is new.
          //
          return (lport);
        }
    }
  //
  // If we are here, no free port was found.
  //
  return ((h_port_t) 0);
}
```

94.5.39 kernel/fs/sock_reference.c

Si veda la sezione 93.6.33.

```
#include <kernel/fs.h>
#include <errno.h>
#include <fcntl.h>
```



```
//---------------------------------------------------------
sock_t *
sock_reference (int skn)
{
  //
  // Check type of request.
  //
  if (skn < 0)
    {
      //
      // Find a free slot.
      //
      for (skn = 0; skn < SOCK_MAX_SLOTS; skn++)
        {
          if (sock_table[skn].active == 0)
            {
              return (&sock_table[skn]);
            }
        }
      return (NULL);
    }
  else if (skn > SOCK_MAX_SLOTS)
    {
      return (NULL);
    }
  else
    {
      return (&sock_table[skn]);
    }
}
```

94.5.40 kernel/fs/zone_alloc.c

Si veda la sezione 93.6.34.

```
#include <kernel/fs.h>
#include <kernel/dev.h>
#include <errno.h>
//---------------------------------------------------------
zno_t
zone_alloc (sb_t * sb)
{
  int m;        // Index inside the inode map.
  int map_element;
  int map_bit;
  int map_mask;
  zno_t zone;
  char buffer[SB_MAX_ZONE_SIZE];
  int status;
  //
  // Verify if write is allowed.
  //
  if (sb->options & MOUNT_RO)
    {
      errset (EROFS);   // Read-only file system.
      return ((zno_t) 0);
    }
  //
  // Write allowed: scan the zone map, to find a free
  // zone. If a free zone can be found, allocate it
  // inside the map.
  // Index 'm' starts from one, because the first bit
  // of the map is reserved for a 'zero' data-zone
  // that does not exist: the second bit is for the
  // real first data-zone.
  //
  for (zone = 0, m = 1; m < (SB_MAP_ZONE_SIZE * 16); m++)
    {
      map_element = m / 16;
      map_bit = m % 16;
      map_mask = 1 << map_bit;
      if (!(sb->map_zone[map_element] & map_mask))
        {
          //
          // Found a free place: set the map.
          //
          sb->map_zone[map_element] |= map_mask;
          sb->changed = 1;
          //
          // The *second* bit inside the map is for
          // the first data zone (the zone after the
          // inode table inside the file system),
          // because the first is for a special
          // 'zero' data zone, not really used.
          //
          zone = sb->first_data_zone + m - 1;   // Found
          // a free
```

```
1150053           // zone.
1150054           //
1150055           // If the zone is outside the disk size, let
1150056           // set the map bit, but reset variable
1150057           // 'zone'.
1150058           //
1150059           if (zone >= sb->zones)
1150060             {
1150061               zone = 0;
1150062             }
1150063           else
1150064             {
1150065               break;
1150066             }
1150067         }
1150068     }
1150069   if (zone == 0)
1150070     {
1150071       errset (ENOSPC);  // No space left on device.
1150072       return ((zno_t) 0);
1150073     }
1150074   //
1150075   // A free zone was found and the map was modified
1150076   // inside
1150077   // the super block in memory. The zone must be
1150078   // cleared.
1150079   //
1150080   status = zone_write (sb, zone, buffer);
1150081   if (status != 0)
1150082     {
1150083       zone_free (sb, zone);
1150084       return ((zno_t) 0);
1150085     }
1150086   //
1150087   // A zone was allocated: return the number.
1150088   //
1150089   return (zone);
1150090 }
```

94.5.41 kernel/fs/zone_free.c

Si veda la sezione 93.6.34.

```
1160001 #include <kernel/fs.h>
1160002 #include <kernel/dev.h>
1160003 #include <errno.h>
1160004 //-----------------------------------------------------------
1160005 int
1160006 zone_free (sb_t * sb, zno_t zone)
1160007 {
1160008   int map_element;
1160009   int map_bit;
1160010   int map_mask;
1160011   //
1160012   // Check arguments.
1160013   //
1160014   if (sb == NULL || zone < sb->first_data_zone)
1160015     {
1160016       errset (EINVAL);  // Invalid argument.
1160017       return (-1);
1160018     }
1160019   //
1160020   // Calculate the map element, the map bit and the
1160021   // map mask.
1160022   //
1160023   // The *second* bit inside the map is for the first
1160024   // data-zone
1160025   // (the zone after the inode table inside the file
1160026   // system),
1160027   // because the first is for a special 'zero'
1160028   // data-zone, not
1160029   // really used.
1160030   //
1160031   map_element = (zone - sb->first_data_zone + 1) / 16;
1160032   map_bit = (zone - sb->first_data_zone + 1) % 16;
1160033   map_mask = 1 << map_bit;
1160034   //
1160035   // Verify if the requested zone is inside the file
1160036   // system area.
1160037   //
1160038   if (zone >= sb->zones)
1160039     {
1160040       errset (EINVAL);  // Invalid argument.
1160041       return (-1);
1160042     }
1160043   //
1160044   // Free the zone and return.
```



```
1160045   //
1160046   if (sb->map_zone[map_element] & map_mask)
1160047     {
1160048       sb->map_zone[map_element] &= ~map_mask;
1160049       sb->changed = 1;
1160050       return (0);
1160051     }
1160052   else
1160053     {
1160054       errset (EUNKNOWN);        // The zone was
1160055       // already free.
1160056       return (-1);
1160057     }
1160058 }
```

94.5.42 kernel/fs/zone_print.c

Si veda la sezione 93.6.36.

```
1170001 #include <sys/os32.h>
1170002 #include <kernel/fs.h>
1170003 #include <kernel/dev.h>
1170004 #include <errno.h>
1170005 //-----------------------------------------------------------
1170006 void
1170007 zone_print (sb_t * sb, zno_t zone)
1170008 {
1170009   char buffer[SB_MAX_ZONE_SIZE];
1170010   int status;
1170011   int i;
1170012   int x;
1170013   //
1170014   status = zone_read (sb, zone, buffer);
1170015   //
1170016   if (status < 0)
1170017     {
1170018       k_perror (NULL);
1170019       return;
1170020     }
1170021   //
1170022   // Print.
1170023   //
1170024   k_printf
1170025     ("dev: 0x%04x, first dzone: %i zone read: %i\n",
1170026      sb->device, sb->first_data_zone, zone);
1170027   //
1170028   // Will print at most the first 256 byte only!
1170029   //
1170030   for (i = 0; i < sb->blksize && i < 256; i++)
1170031     {
1170032       k_printf ("%02x ", buffer[i]);
1170033       x = (i + 1) % 4;
1170034       if (x == 0 && i > 0)
1170035         {
1170036           k_printf ("| ");
1170037         }
1170038       x = (i + 1) % 16;
1170039       if (x == 0 && i > 0)
1170040         {
1170041           k_printf ("\n");
1170042         }
1170043     }
1170044 }
```

94.5.43 kernel/fs/zone_read.c

Si veda la sezione 93.6.37.

```
1180001 #include <sys/os32.h>
1180002 #include <kernel/fs.h>
1180003 #include <kernel/dev.h>
1180004 #include <errno.h>
1180005 //-----------------------------------------------------------
1180006 int
1180007 zone_read (sb_t * sb, zno_t zone, void *buffer)
1180008 {
1180009   size_t size_zone;
1180010   off_t off_start;
1180011   ssize_t size_read;
1180012   //
1180013   // Verify if the requested zone is inside the file
1180014   // system area.
1180015   //
1180016   if (zone >= sb->zones)
1180017     {
1180018       errset (EINVAL);  // Invalid argument.
1180019       return (-1);
```

```
1180020     }
1180021   //
1180022   // Calculate start position.
1180023   //
1180024   size_zone = 1024 << sb->log2_size_zone;
1180025   off_start = zone;
1180026   off_start *= size_zone;
1180027   //
1180028   // Read from device to the buffer.
1180029   //
1180030   size_read =
1180031     dev_io ((pid_t) - 1, sb->device, DEV_READ,
1180032             off_start, buffer, size_zone, NULL);
1180033   if (size_read != size_zone)
1180034     {
1180035       errset (EIO);     // I/O error.
1180036       return (-1);
1180037     }
1180038   else
1180039     {
1180040       return (0);
1180041     }
1180042 }
```

94.5.44 kernel/fs/zone_write.c

«

Si veda la sezione 93.6.37.

```
1190001 #include <kernel/fs.h>
1190002 #include <kernel/dev.h>
1190003 #include <errno.h>
1190004 //-----------------------------------------------------------
1190005 int
1190006 zone_write (sb_t * sb, zno_t zone, void *buffer)
1190007 {
1190008   size_t size_zone;
1190009   off_t off_start;
1190010   ssize_t size_written;
1190011   //
1190012   // Verify if write is allowed.
1190013   //
1190014   if (sb->options & MOUNT_RO)
1190015     {
1190016       errset (EROFS);   // Read-only file system.
1190017       return (-1);
1190018     }
1190019   //
1190020   // Verify if the requested zone is inside the file
1190021   // system area.
1190022   //
1190023   if (zone >= sb->zones)
1190024     {
1190025       errset (EINVAL);  // Invalid argument.
1190026       return (-1);
1190027     }
1190028   //
1190029   // Write is allowed: calculate start position.
1190030   //
1190031   size_zone = 1024 << sb->log2_size_zone;
1190032   off_start = zone;
1190033   off_start *= size_zone;
1190034   //
1190035   // Write the buffer to the device.
1190036   //
1190037   size_written =
1190038     dev_io ((pid_t) - 1, sb->device, DEV_WRITE,
1190039             off_start, buffer, size_zone, NULL);
1190040   if (size_written != size_zone)
1190041     {
1190042       errset (EIO);     // I/O error.
1190043       return (-1);
1190044     }
1190045   else
1190046     {
1190047       return (0);
1190048     }
1190049 }
```

94.6 os32: «kernel/ibm_i386.h»

«

Si veda la sezione 93.7.

```
1200001 #ifndef _KERNEL_IBM_I386_H
1200002 #define _KERNEL_IBM_I386_H   1
1200003 //-----------------------------------------------------------
1200004 #include <stdint.h>
1200005 #include <inttypes.h>
```



```
1200006 #include <stdbool.h>
1200007 #include <stdarg.h>
1200008 //-----------------------------------------------------------
1200009 // GDT
1200010 //-----------------------------------------------------------
1200011 #define GDT_ITEMS       256     // Max is 8192 items.
1200012 //
1200013 typedef struct
1200014 {
1200015   uint32_t limit_a:16, base_a:16;
1200016   uint32_t base_b:8,
1200017     accessed:1,
1200018     write_execute:1,
1200019     expansion_conforming:1,
1200020     code_or_data:1,
1200021     code_data_or_system:1,
1200022     dpl:2,
1200023     present:1,
1200024     limit_b:4,
1200025     available:1, reserved:1, big:1, granularity:1, base_c:8;
1200026 } gdt_t;
1200027 //
1200028 extern gdt_t gdt_table[GDT_ITEMS];
1200029 //-----------------------------------------------------------
1200030 typedef struct
1200031 {
1200032   uint16_t limit;
1200033   uint32_t base;
1200034 } __attribute__ ((packed)) gdtr_t;      // [1]
1200035 //
1200036 extern gdtr_t gdt_register;
1200037 //
1200038 // [1] It is necessary that the structure be compact,
1200039 //     so that it uses exactly 48 bits. That is why the
1200040 //     attribute 'packed' for the GNU C compiler is
1200041 //     used.
1200042 //-----------------------------------------------------------
1200043 int gdt_segment (int segment, uint32_t base,
1200044                  uint32_t limit, bool present,
1200045                  bool code, unsigned char dpl);
1200046 //
1200047 void gdt_print (void *gdtr, unsigned int first,
1200048                 unsigned int last);
1200049 void gdt_load (void *gdtr);
1200050 void gdt (void);
1200051 //
1200052 // Segment 0 is not used,
1200053 // segment 1 is for kernel code,
1200054 // segment 2 is for kernel data,
1200055 // segment 3 is for process 1 code,
1200056 // segment 4 is for process 1 data,
1200057 // ...
1200058 //
1200059 #define gdt_pid_to_segment_text(p)  (p*2+1)
1200060 #define gdt_pid_to_segment_data(p)  (p*2+2)
1200061 #define gdt_segment_text_to_pid(s)  (s/2)
1200062 #define gdt_segment_data_to_pid(s)  (s/2-1)
1200063 //-----------------------------------------------------------
1200064 // IDT
1200065 //-----------------------------------------------------------
1200066 #define IDT_ITEMS      129      // 0-128 0x00-0x80
1200067 //-----------------------------------------------------------
1200068 typedef struct
1200069 {
1200070   uint32_t offset_a:16, selector:16;
1200071   uint32_t filler:8,
1200072     type:4, system:1, dpl:2, present:1, offset_b:16;
1200073 } idt_t;
1200074 //
1200075 extern idt_t idt_table[IDT_ITEMS];
1200076 //-----------------------------------------------------------
1200077 typedef struct
1200078 {
1200079   uint16_t limit;
1200080   uint32_t base;
1200081 } __attribute__ ((packed)) idtr_t;
1200082 //
1200083 extern idtr_t idt_register;
1200084 //-----------------------------------------------------------
1200085 void idt_descriptor (int desc, void *isr,
1200086                      uint16_t selector, bool present,
1200087                      char type, char dpl);
1200088 void idt_load (void *idtr);
1200089 void idt (void);
1200090 void idt_irq_remap (unsigned int offset_1,
1200091                     unsigned int offset_2);
1200092 void idt_print (void *idtr, unsigned int first,
```

```
1200093                 unsigned int last);
1200094 //-------------------------------------------------------
1200095 // ISR
1200096 //-------------------------------------------------------
1200097 void isr_0 (void);
1200098 void isr_1 (void);
1200099 void isr_2 (void);
1200100 void isr_3 (void);
1200101 void isr_4 (void);
1200102 void isr_5 (void);
1200103 void isr_6 (void);
1200104 void isr_7 (void);
1200105 void isr_8 (void);
1200106 void isr_9 (void);
1200107 void isr_10 (void);
1200108 void isr_11 (void);
1200109 void isr_12 (void);
1200110 void isr_13 (void);
1200111 void isr_14 (void);
1200112 void isr_15 (void);
1200113 void isr_16 (void);
1200114 void isr_17 (void);
1200115 void isr_18 (void);
1200116 void isr_19 (void);
1200117 void isr_20 (void);
1200118 void isr_21 (void);
1200119 void isr_22 (void);
1200120 void isr_23 (void);
1200121 void isr_24 (void);
1200122 void isr_25 (void);
1200123 void isr_26 (void);
1200124 void isr_27 (void);
1200125 void isr_28 (void);
1200126 void isr_29 (void);
1200127 void isr_30 (void);
1200128 void isr_31 (void);
1200129 void isr_32 (void);
1200130 void isr_33 (void);
1200131 void isr_34 (void);
1200132 void isr_35 (void);
1200133 void isr_36 (void);
1200134 void isr_37 (void);
1200135 void isr_38 (void);
1200136 void isr_39 (void);
1200137 void isr_40 (void);
1200138 void isr_41 (void);
1200139 void isr_42 (void);
1200140 void isr_43 (void);
1200141 void isr_44 (void);
1200142 void isr_45 (void);
1200143 void isr_46 (void);
1200144 void isr_47 (void);
1200145 void isr_128 (void);
1200146 //
1200147 char *isr_exception_name (int exception);
1200148 //
1200149 void isr_exception_unrecoverable (uint32_t eax,
1200150                                   uint32_t ecx,
1200151                                   uint32_t edx,
1200152                                   uint32_t ebx,
1200153                                   uint32_t ebp,
1200154                                   uint32_t esi,
1200155                                   uint32_t edi,
1200156                                   uint32_t ds,
1200157                                   uint32_t es,
1200158                                   uint32_t fs,
1200159                                   uint32_t gs,
1200160                                   uint32_t interrupt,
1200161                                   uint32_t error,
1200162                                   uint32_t eip,
1200163                                   uint32_t cs,
1200164                                   uint32_t eflags);
1200165 //
1200166 void isr_irq_clear (uint32_t idtn);
1200167 void isr_irq_clear_pic1 (void);
1200168 void isr_irq_clear_pic2 (void);
1200169 //-------------------------------------------------------
1200170 // I/O
1200171 //-------------------------------------------------------
1200172 uint32_t _in_8 (uint32_t port);
1200173 uint32_t _in_16 (uint32_t port);
1200174 uint32_t _in_32 (uint32_t port);
1200175 void _out_8 (uint32_t port, uint32_t value);
1200176 void _out_16 (uint32_t port, uint32_t value);
1200177 void _out_32 (uint32_t port, uint32_t value);
1200178 //
1200179 #define in_8(port)  \
```



```
1200180     ((unsigned int) _in_8 ((uint32_t) port))
1200181 #define in_16(port) \
1200182     ((unsigned int) _in_16 ((uint32_t) port))
1200183 #define in_32(port) \
1200184     ((unsigned int) _in_32 ((uint32_t) port))
1200185 #define out_8(port, value) \
1200186     (_out_8 ((uint32_t) port, (uint32_t) value))
1200187 #define out_16(port, value) \
1200188     (_out_16 ((uint32_t) port, (uint32_t) value))
1200189 #define out_32(port, value) \
1200190     (_out_32 ((uint32_t) port, (uint32_t) value))
1200191 //-------------------------------------------------------
1200192 // Interrupt on/off
1200193 //-------------------------------------------------------
1200194 void cli (void);
1200195 void sti (void);
1200196 void irq_on (unsigned int irq);
1200197 void irq_off (unsigned int irq);
1200198 //-------------------------------------------------------
1200199 #endif
```


94.6.1 kernel/ibm_i386/_in_16.s

Si veda la sezione 93.7.

```
1210001 .global _in_16
1210002 #---------------------------------------------------
1210003 .section .text
1210004 #---------------------------------------------------
1210005 # Port input word.
1210006 #---------------------------------------------------
1210007 _in_16:
1210008     enter $4, $0                # 1 local variable.
1210009     pushf
1210010     pusha
1210011     .equ IN_PORT,  8            # First argument.
1210012     .equ IN_DATA, -4            # Local variable.
1210013     mov  IN_PORT(%ebp), %edx    # Copy the port number
1210014                                 # into EDX, but
1210015                                 # then only DX will be
```

```
1210016                                 # used.
1210017     mov   $0, %eax              # Reset EAX.
1210018     in    %dx,  %ax             # Read DX port  and
1210019                                 # save into AX.
1210020     mov   %eax, IN_DATA(%ebp)   # Save EAX inside the
1210021                                 # local variable.
1210022     popa
1210023     popf
1210024     mov   IN_DATA(%ebp), %eax   # Restore EAX and
1210025     leave                       # return.
1210026     ret
```

94.6.2 kernel/ibm_i386/_in_32.s

Si veda la sezione 93.7.

```
1220001 .global _in_32
1220002 #-------------------------------------------------
1220003 .section .text
1220004 #-------------------------------------------------
1220005 # Port input word.
1220006 #-------------------------------------------------
1220007 _in_32:
1220008     enter $4, $0                # 1 local variable.
1220009     pushf
1220010     pusha
1220011     .equ IN_PORT,  8            # First argument.
1220012     .equ IN_DATA, -4            # Local variable.
1220013     mov  IN_PORT(%ebp), %edx    # Copy the port number
1220014                                 # into EDX, but
1220015                                 # then only DX will be
1220016                                 # used.
1220017     mov   $0, %eax              # Reset EAX.
1220018     inl   %dx, %eax             # Read DX port  and
1220019                                 # save into EAX.
1220020     mov   %eax, IN_DATA(%ebp)   # Save EAX inside the
1220021                                 # local variable.
1220022     popa
1220023     popf
1220024     mov   IN_DATA(%ebp), %eax   # Restore EAX and
1220025     leave                       # return.
1220026     ret
```

94.6.3 kernel/ibm_i386/_in_8.s

Si veda la sezione 93.7.

```
1230001 .global _in_8
1230002 #-------------------------------------------------
1230003 .section .text
1230004 #-------------------------------------------------
1230005 # Port input byte.
1230006 #-------------------------------------------------
1230007 _in_8:
1230008     enter $4, $0                # 1 local variable.
1230009     pushf
1230010     pusha
1230011     .equ IN_PORT,  8            # First argument.
1230012     .equ IN_DATA, -4            # Local variable.
1230013     mov  IN_PORT(%ebp), %edx    # Copy the port number
1230014                                 # into EDX, but
1230015                                 # then only DX will be
1230016                                 # used.
1230017     mov   $0, %eax              # Reset EAX.
1230018     inb   %dx,  %al             # Read DX port  and
1230019                                 # save into AL.
1230020     mov   %eax, IN_DATA(%ebp)   # Save EAX inside the
1230021                                 # local variable.
1230022     popa
1230023     popf
1230024     mov   IN_DATA(%ebp), %eax   # Restore EAX and
1230025     leave                       # return.
1230026     ret
```

94.6.4 kernel/ibm_i386/_out_16.s

Si veda la sezione 93.7.

```
1240001 .global _out_16
1240002 #-------------------------------------------------
1240003 .section .text
1240004 #-------------------------------------------------
1240005 # Port output word.
1240006 #-------------------------------------------------
1240007 _out_16:
1240008     enter $0, $0                # No local variables.
1240009     pushf
```



```
1240010     pusha
1240011     .equ  OUT_PORT,  8          # First parameter.
1240012     .equ  OUT_DATA, 12          # Second parameter.
1240013     mov   OUT_PORT(%ebp), %edx  # Copy output port to
1240014                                 # EDX, but only DX
1240015                                 # will be used.
1240016     mov   OUT_DATA(%ebp), %eax  # Copy output data to
1240017                                 # EAX, but only AX
1240018                                 # will be used.
1240019     out   %ax, %dx              # Send to the port.
1240020     popa
1240021     popf
1240022     leave
1240023     ret
```

94.6.5 kernel/ibm_i386/_out_32.s

Si veda la sezione 93.7.

```
1250001 .global _out_32
1250002 #-------------------------------------------------
1250003 .section .text
1250004 #-------------------------------------------------
1250005 # Port output word.
1250006 #-------------------------------------------------
1250007 _out_32:
1250008     enter $0, $0                # No local variables.
1250009     pushf
1250010     pusha
1250011     .equ  OUT_PORT,  8          # First parameter.
1250012     .equ  OUT_DATA, 12          # Second parameter.
1250013     mov   OUT_PORT(%ebp), %edx  # Copy output port to
1250014                                 # EDX, but only DX
1250015                                 # will be used.
1250016     mov   OUT_DATA(%ebp), %eax  # Copy output data to
1250017                                 # EAX.
1250018     outl  %eax, %dx             # Send to the port.
1250019     popa
1250020     popf
1250021     leave
1250022     ret
```

94.6.6 kernel/ibm_i386/_out_8.s

Si veda la sezione 93.7.

```
1260001 .global _out_8
1260002 #-------------------------------------------------
1260003 .section .text
1260004 #-------------------------------------------------
1260005 # Port output byte.
1260006 #-------------------------------------------------
1260007 _out_8:
1260008     enter $0, $0                # No local variables.
1260009     pushf
1260010     pusha
1260011     .equ  OUT_PORT,  8          # First parameter.
1260012     .equ  OUT_DATA, 12          # Second parameter.
1260013     mov   OUT_PORT(%ebp), %edx  # Copy output port to
1260014                                 # EDX, but only DX
1260015                                 # will be used.
1260016     mov   OUT_DATA(%ebp), %eax  # Copy output data to
1260017                                 # EAX, but only AL
1260018                                 # will be used.
1260019     outb  %al, %dx              # Send to the port.
1260020     popa
1260021     popf
1260022     leave
1260023     ret
```

94.6.7 kernel/ibm_i386/cli.s

Si veda la sezione 93.7.

```
1270001 .global cli
1270002 #-------------------------------------------------
1270003 .text
1270004 #-------------------------------------------------
1270005 # Clear interrupt flag.
1270006 #-------------------------------------------------
1270007 .align 4
1270008 cli:
1270009     cli
1270010     ret
```

94.6.8 kernel/ibm_i386/gdt.c

Si veda la sezione 93.7.

```
#include <kernel/ibm_i386.h>
#include <kernel/multiboot.h>
//-------------------------------------------------------
void
gdt (void)
{
  uint32_t blocks;      // Total available memory
  // blocks.
  //
  // Calculate memory blocks.
  //
  blocks = (multiboot.mem_upper * 1024) / 4096;
  //
  // Set data for GDTR register.
  //
  gdt_register.limit = (sizeof (gdt_table) - 1);
  gdt_register.base = (uint32_t) & gdt_table[0];
  //
  // Reset items inside 'gdt_table[]'.
  // gdt_table[0] must be null and is not to be
  // used.
  //
  int i;
  for (i = 0; i < GDT_ITEMS; i++)
    {
      gdt_segment (i, 0, 0, 0, 0, 0);
    }
  //
  // gdt_table[1] is for kernel code.
  // It covers all the available memory, with DPL 0.
  // The selector is 8+0 = 0x08+0.
  //
  gdt_segment (1, 0, blocks, 1, 1, 0);
  //
  // gdt_table[2] is for kernel data, including stack
  // (the stack
  // MUST be in the same address space of data).
  // It covers all the available memory, with DPL 0.
  // The selector is 16+0 = 0x10+0.
  //
  gdt_segment (2, 0, blocks, 1, 0, 0);
  //
  // Load the GDT table.
  //
  gdt_load (&gdt_register);
}
```

94.6.9 kernel/ibm_i386/gdt_load.s

Si veda la sezione 93.7.

```
.globl  gdt_load
#
gdt_load:
    enter $0, $0
    .equ gdtr_pointer,  8          # Primo argomento.
    mov  gdtr_pointer(%ebp), %eax  # Copia il
                                   # puntatore in EAX.
    leave
    #
    lgdt (%eax)         # Carica il registro GDTR
                        # dall'indirizzo in EAX.
    #
    # 2 kernel data, included stack, DPL 0, covering
    #   all the available
    #   memory: segment selector 0x10+0.
    #
    mov  $16, %ax
    mov  %ax, %ds
    mov  %ax, %es
    mov  %ax, %fs
    mov  %ax, %gs
    mov  %ax, %ss       # The stack MUST be in the same
                        # address space of the other
                        # data, to allow pointers to
    #                   # work correctly.
    #
    # 2 kernel code, DPL 0, covering all the available
    #   memory:
    #   segment selector 0x08+0.
    #
    jmp  $8, $flush
flush:
    ret
```


94.6.10 kernel/ibm_i386/gdt_print.c

Si veda la sezione 93.7.

```
#include <kernel/ibm_i386.h>
#include <kernel/lib_k.h>
//
void
gdt_print (void *gdtr, unsigned int first,
           unsigned int last)
{
  gdtr_t *g = gdtr;
  uint32_t *p = (uint32_t *) g->base;
  //
  int max = (g->limit + 1) / (sizeof (uint32_t));
  int i;
  //
  if (((first * 2) > max) || (first > last))
    {
      return;
    }
  //
  k_printf ("[%s] base: 0x%08" PRIX32 " limit: 0x%04"
            PRIX32 "\n", __func__, g->base, g->limit);
  //
  for (i = (first * 2); i < max && i <= (last * 2); i += 2)
    {
      k_printf ("[%4" PRIx32 "] %032" PRIb32 " %032"
                PRIb32 "\n", i / 2, p[i], p[i + 1]);
    }
}
```

94.6.11 kernel/ibm_i386/gdt_public.c

Si veda la sezione 93.7.

```
#include <kernel/ibm_i386.h>
//-------------------------------------------------------
gdt_t gdt_table[GDT_ITEMS];
gdtr_t gdt_register;
```

94.6.12 kernel/ibm_i386/gdt_segment.c

Si veda la sezione 93.7.

```
#include <kernel/ibm_i386.h>
#include <errno.h>
//-------------------------------------------------------
int
gdt_segment (int segment,
             uint32_t base,
             uint32_t limit,
             bool present, bool code, unsigned char dpl)
{
  //
  // Verify if the segment is valid.
  //
  if ((segment >= ((sizeof (gdt_table)) / 8))
      || (segment < 0))
    {
      errset (EINVAL);
      return (-1);
    }
  //
  // Limit.
  //
  gdt_table[segment].limit_a = (limit & 0x0000FFFF);
  gdt_table[segment].limit_b = limit / 0x10000;
  //
  // Base address.
  //
  gdt_table[segment].base_a = (base & 0x0000FFFF);
  gdt_table[segment].base_b =
    ((base / 0x10000) & 0x000000FF);
  gdt_table[segment].base_c = (base / 0x1000000);
  //
  // Attributes.
  //

  //
  // Internal update.
  //
  gdt_table[segment].accessed = 0;
  //
  // r, w, x.
  //
  gdt_table[segment].write_execute = 1;
  //
```

```
1320044   // Normal and conforming.
1320045   //
1320046   gdt_table[segment].expansion_conforming = 0;
1320047   //
1320048   gdt_table[segment].code_or_data = code;
1320049   gdt_table[segment].code_data_or_system = 1;
1320050   gdt_table[segment].dpl = dpl;
1320051   gdt_table[segment].present = present;
1320052   gdt_table[segment].available = 0;     // 0
1320053   gdt_table[segment].reserved = 0;      // 0
1320054   gdt_table[segment].big = 1;   // 32 bit
1320055   gdt_table[segment].granularity = 1;   // 4 Kibyte
1320056   //
1320057   return (0);
1320058 }
```

94.6.13 kernel/ibm_i386/idt.c

«

Si veda la sezione 93.7.

```
1330001 #include <kernel/ibm_i386.h>
1330002 //-----------------------------------------------------------
1330003 void
1330004 idt (void)
1330005 {
1330006   //
1330007   // Set necessary data for the IDTR register.
1330008   //
1330009   idt_register.limit = (sizeof (idt_table) - 1);
1330010   idt_register.base = (uint32_t) & idt_table[0];
1330011   //
1330012   // Reset all items inside the array 'idt_table[]'.
1330013   //
1330014   int i;
1330015   for (i = 0; i < IDT_ITEMS; i++)
1330016     {
1330017       idt_descriptor (i, 0, 0, 0, 0, 0);
1330018     }
1330019   //
1330020   // Place hardware interrupt from IRQ 0 to IRQ 7
1330021   // starting from descriptor 32 and from IRQ 8 to
1330022   // IRQ 15 starting from descriptor 40.
1330023   //
1330024   idt_irq_remap (32, 40);
1330025   //
1330026   // Set the ISR routines to the items inside the IDT
1330027   // table.
1330028   //
1330029   idt_descriptor (0, isr_0, 0x0008, 1, 0xE, 0);
1330030   idt_descriptor (1, isr_1, 0x0008, 1, 0xE, 0);
1330031   idt_descriptor (2, isr_2, 0x0008, 1, 0xE, 0);
1330032   idt_descriptor (3, isr_3, 0x0008, 1, 0xE, 0);
1330033   idt_descriptor (4, isr_4, 0x0008, 1, 0xE, 0);
1330034   idt_descriptor (5, isr_5, 0x0008, 1, 0xE, 0);
1330035   idt_descriptor (6, isr_6, 0x0008, 1, 0xE, 0);
1330036   idt_descriptor (7, isr_7, 0x0008, 1, 0xE, 0);
1330037   idt_descriptor (8, isr_8, 0x0008, 1, 0xE, 0);
1330038   idt_descriptor (9, isr_9, 0x0008, 1, 0xE, 0);
1330039   idt_descriptor (10, isr_10, 0x0008, 1, 0xE, 0);
1330040   idt_descriptor (11, isr_11, 0x0008, 1, 0xE, 0);
1330041   idt_descriptor (12, isr_12, 0x0008, 1, 0xE, 0);
1330042   idt_descriptor (13, isr_13, 0x0008, 1, 0xE, 0);
1330043   idt_descriptor (14, isr_14, 0x0008, 1, 0xE, 0);
1330044   idt_descriptor (15, isr_15, 0x0008, 1, 0xE, 0);
1330045   idt_descriptor (16, isr_16, 0x0008, 1, 0xE, 0);
1330046   idt_descriptor (17, isr_17, 0x0008, 1, 0xE, 0);
1330047   idt_descriptor (18, isr_18, 0x0008, 1, 0xE, 0);
1330048   idt_descriptor (19, isr_19, 0x0008, 1, 0xE, 0);
1330049   idt_descriptor (20, isr_20, 0x0008, 1, 0xE, 0);
1330050   idt_descriptor (21, isr_21, 0x0008, 1, 0xE, 0);
1330051   idt_descriptor (22, isr_22, 0x0008, 1, 0xE, 0);
1330052   idt_descriptor (23, isr_23, 0x0008, 1, 0xE, 0);
1330053   idt_descriptor (24, isr_24, 0x0008, 1, 0xE, 0);
1330054   idt_descriptor (25, isr_25, 0x0008, 1, 0xE, 0);
1330055   idt_descriptor (26, isr_26, 0x0008, 1, 0xE, 0);
1330056   idt_descriptor (27, isr_27, 0x0008, 1, 0xE, 0);
1330057   idt_descriptor (28, isr_28, 0x0008, 1, 0xE, 0);
1330058   idt_descriptor (29, isr_29, 0x0008, 1, 0xE, 0);
1330059   idt_descriptor (30, isr_10, 0x0008, 1, 0xE, 0);
1330060   idt_descriptor (31, isr_31, 0x0008, 1, 0xE, 0);
1330061   idt_descriptor (32, isr_32, 0x0008, 1, 0xE, 0);
1330062   idt_descriptor (33, isr_33, 0x0008, 1, 0xE, 0);
1330063   idt_descriptor (34, isr_34, 0x0008, 1, 0xE, 0);
1330064   idt_descriptor (35, isr_35, 0x0008, 1, 0xE, 0);
1330065   idt_descriptor (36, isr_36, 0x0008, 1, 0xE, 0);
1330066   idt_descriptor (37, isr_37, 0x0008, 1, 0xE, 0);
1330067   idt_descriptor (38, isr_38, 0x0008, 1, 0xE, 0);
```



```
1330068   idt_descriptor (39, isr_39, 0x0008, 1, 0xE, 0);
1330069   idt_descriptor (40, isr_40, 0x0008, 1, 0xE, 0);
1330070   idt_descriptor (41, isr_41, 0x0008, 1, 0xE, 0);
1330071   idt_descriptor (42, isr_42, 0x0008, 1, 0xE, 0);
1330072   idt_descriptor (43, isr_43, 0x0008, 1, 0xE, 0);
1330073   idt_descriptor (44, isr_44, 0x0008, 1, 0xE, 0);
1330074   idt_descriptor (45, isr_45, 0x0008, 1, 0xE, 0);
1330075   idt_descriptor (46, isr_46, 0x0008, 1, 0xE, 0);
1330076   idt_descriptor (47, isr_47, 0x0008, 1, 0xE, 0);
1330077   //
1330078   // The following item is for the system calls.
1330079   //
1330080   idt_descriptor (128, isr_128, 0x0008, 1, 0xE, 0);
1330081   //
1330082   // Activate the IDT loading the IDTR register.
1330083   //
1330084   idt_load (&idt_register);
1330085 }
```

94.6.14 kernel/ibm_i386/idt_descriptor.c

«

Si veda la sezione 93.7.

```
1340001 #include <kernel/ibm_i386.h>
1340002 //-----------------------------------------------------------
1340003 void
1340004 idt_descriptor (int desc,
1340005                 void *isr,
1340006                 uint16_t selector,
1340007                 bool present, char type, char dpl)
1340008 {
1340009   uint32_t offset = (uint32_t) isr;
1340010   //
1340011   // Unset reserved bits and the system bit.
1340012   //
1340013   idt_table[desc].filler = 0;
1340014   idt_table[desc].system = 0;
1340015   //
1340016   // Relative address.
1340017   //
1340018   idt_table[desc].offset_a = (offset & 0x0000FFFF);
1340019   idt_table[desc].offset_b = (offset / 0x10000);
1340020   //
1340021   // Selector.
1340022   //
1340023   idt_table[desc].selector = selector;
1340024   //
1340025   // Valid item?
1340026   //
1340027   idt_table[desc].present = present;
1340028   //
1340029   // Type (gate type).
1340030   //
1340031   idt_table[desc].type = (type & 0x0F);
1340032   //
1340033   // DPL.
1340034   //
1340035   idt_table[desc].dpl = (dpl & 0x03);
1340036 }
```

94.6.15 kernel/ibm_i386/idt_irq_remap.c

«

Si veda la sezione 93.7.

```
1350001 #include <kernel/lib_k.h>
1350002 #include <kernel/ibm_i386.h>
1350003 //-----------------------------------------------------------
1350004 #define DEBUG 0
1350005 //-----------------------------------------------------------
1350006 void
1350007 idt_irq_remap (unsigned int offset_1, unsigned int offset_2)
1350008 {
1350009   //
1350010   // PIC_P è il PIC primario o «master»;
1350011   // PIC_S è il PIC secondario o «slave».
1350012   //
1350013   // Quando si manifesta un IRQ che riguarda il PIC
1350014   // secondario,
1350015   // il PIC primario riceve IRQ 2.
1350016   //
1350017   // ICW = initialization command word.
1350018   // OCW = operation command word.
1350019   //
1350020   if (DEBUG)
1350021     {
1350022       k_printf
1350023         ("[%s] PIC (programmable interrupt "
```

```
1350024           "controller) " "remap: ", __func__);
1350025       }
1350026     //
1350027     out_8 (0x20, 0x10 + 0x01);    // Initialization:
1350028     // 0x10 means that is
1350029     out_8 (0xA0, 0x10 + 0x01);    // is ICW1; 0x01 means
1350030     // that must
1350031     // continue up to ICW4.
1350032     if (DEBUG)
1350033       {
1350034         k_printf ("ICW1");
1350035       }
1350036     out_8 (0x21, offset_1);       // ICW2: PIC_P
1350037     // starting at
1350038     // «offset_1».
1350039     out_8 (0xA1, offset_2);       // PIC_S starting at
1350040     // «offset_2».
1350041     if (DEBUG)
1350042       {
1350043         k_printf (", ICW2");
1350044       }
1350045     out_8 (0x21, 0x04);   // ICW3 PIC_P: IRQ2 driven
1350046     // from PIC_S.
1350047     out_8 (0xA1, 0x02);   // ICW3 PIC_S: driving IRQ2
1350048     // from PIC_P.
1350049     if (DEBUG)
1350050       {
1350051         k_printf (", ICW3");
1350052       }
1350053     out_8 (0x21, 0x01);   // ICW4: si precisa solo la
1350054     // modalità
1350055     out_8 (0xA1, 0x01);   // del microprocessore; 0x01 =
1350056     // 8086.
1350057     if (DEBUG)
1350058       {
1350059         k_printf (", ICW4");
1350060       }
1350061     out_8 (0x21, 0x00);   // OCW1: reset mask to enable
1350062     // all
1350063     out_8 (0xA1, 0x00);   // IRQ numbers.
1350064     if (DEBUG)
1350065       {
1350066         k_printf (", OCW1.\n");
1350067       }
1350068 }
```

94.6.16 kernel/ibm_i386/idt_load.s

«

Si veda la sezione 93.7.

```
1360001 .globl  idt_load
1360002 #
1360003 idt_load:
1360004     enter $0, $0
1360005     .equ idtr_pointer,  8          # Primo argomento.
1360006     mov  idtr_pointer(%ebp), %eax  # Copia il puntatore
1360007                                    # in EAX.
1360008     leave
1360009     #
1360010     lidt (%eax)     # Utilizza la tabella IDT a cui
1360011                     # punta EAX.
1360012     #
1360013     ret
```

94.6.17 kernel/ibm_i386/idt_print.c

«

Si veda la sezione 93.7.

```
1370001 #include <kernel/ibm_i386.h>
1370002 #include <kernel/lib_k.h>
1370003 //
1370004 void
1370005 idt_print (void *idtr, unsigned int first,
1370006            unsigned int last)
1370007 {
1370008   idtr_t *g = idtr;
1370009   uint32_t *p = (uint32_t *) g->base;
1370010   //
1370011   int max = (g->limit + 1) / (sizeof (uint32_t));
1370012   int i;
1370013   //
1370014   if (((first * 2) > max) || (first > last))
1370015     {
1370016       return;
1370017     }
1370018   //
1370019   k_printf ("[%s] base: 0x%08" PRIX32 " limit: 0x%04"
1370020             PRIX32 "\n", __func__, g->base, g->limit);
1370021   //
1370022   for (i = (first * 2); i < max && i <= (last * 2); i += 2)
1370023     {
1370024       k_printf ("[%4" PRIx32 "] %032" PRIb32 " %032"
1370025                 PRIb32 "\n", i / 2, p[i], p[i + 1]);
1370026     }
1370027 }
```


94.6.18 kernel/ibm_i386/idt_public.c

«

Si veda la sezione 93.7.

```
1380001 #include <kernel/ibm_i386.h>
1380002 //-------------------------------------------------------------
1380003 idt_t idt_table[129];
1380004 idtr_t idt_register;
```

94.6.19 kernel/ibm_i386/irq_off.c

«

Si veda la sezione 93.7.

```
1390001 #include <kernel/ibm_i386.h>
1390002 #include <stdint.h>
1390003 //-------------------------------------------------------------
1390004 void
1390005 irq_off (unsigned int irq)
1390006 {
1390007   uint32_t mask;
1390008   uint32_t status;
1390009   //
1390010   if (irq > 15)
1390011     {
1390012       return;   // There is not such IRQ.
1390013     }
1390014   else
1390015     {
1390016       mask = ((uint32_t) 1 << irq);
1390017       //
1390018       // IRQ from 0 to 7.
1390019       //
1390020       status = in_8 ((uint32_t) 0x21);
1390021       status = status | mask;
1390022       out_8 ((uint32_t) 0x21, status);
1390023       //
1390024       // IRQ from 8 to 15.
1390025       //
1390026       status = in_8 ((uint32_t) 0xA1);
1390027       status = status | (mask >> 8);
1390028       out_8 ((uint32_t) 0xA1, status);
1390029     }
1390030 }
```

94.6.20 kernel/ibm_i386/irq_on.c

«

Si veda la sezione 93.7.

```
1400001 #include <kernel/ibm_i386.h>
1400002 #include <stdint.h>
1400003 //-------------------------------------------------------------
1400004 void
1400005 irq_on (unsigned int irq)
1400006 {
1400007   uint32_t mask;
1400008   uint32_t status;
1400009   //
1400010   if (irq > 15)
1400011     {
1400012       return;   // There is not such IRQ.
1400013     }
1400014   else
1400015     {
1400016       mask = ~((uint32_t) 1 << irq);
1400017       //
1400018       // IRQ from 0 to 7.
1400019       //
1400020       status = in_8 ((uint32_t) 0x21);
1400021       status = status & mask;
1400022       out_8 ((uint32_t) 0x21, status);
1400023       //
1400024       // IRQ from 8 to 15.
1400025       //
1400026       status = in_8 ((uint32_t) 0xA1);
1400027       status = status & (mask >> 8);
1400028       out_8 ((uint32_t) 0xA1, status);
1400029     }
1400030 }
```

94.6.21 kernel/ibm_i386/isr.s

«

Si veda la sezione 93.7.

```
1410001  .extern isr_exception_unrecoverable
1410002  .extern isr_irq_clear
1410003  .extern isr_irq_clear_pic1
1410004  .extern isr_irq_clear_pic2
1410005  .extern kbd_isr
1410006  .extern sysroutine
1410007  .extern proc_current
1410008  .extern proc_stack_segment_selector;
1410009  .extern proc_stack_pointer
1410010  .extern proc_scheduler
1410011 #.extern proc_sch_terminals
1410012 #
1410013  .global _clock_kernel
1410014  .global _clock_time
1410015  .global _ksp
1410016 #
1410017  .global isr_0
1410018  .global isr_1
1410019  .global isr_2
1410020  .global isr_3
1410021  .global isr_4
1410022  .global isr_5
1410023  .global isr_6
1410024  .global isr_7
1410025  .global isr_8
1410026  .global isr_9
1410027  .global isr_10
1410028  .global isr_11
1410029  .global isr_12
1410030  .global isr_13
1410031  .global isr_14
1410032  .global isr_15
1410033  .global isr_16
1410034  .global isr_17
1410035  .global isr_18
1410036  .global isr_19
1410037  .global isr_20
1410038  .global isr_21
1410039  .global isr_22
1410040  .global isr_23
1410041  .global isr_24
1410042  .global isr_25
1410043  .global isr_26
1410044  .global isr_27
1410045  .global isr_28
1410046  .global isr_29
1410047  .global isr_30
1410048  .global isr_31
1410049  .global isr_32
1410050  .global isr_33
1410051  .global isr_34
1410052  .global isr_35
1410053  .global isr_36
1410054  .global isr_37
1410055  .global isr_38
1410056  .global isr_39
1410057  .global isr_40
1410058  .global isr_41
1410059  .global isr_42
1410060  .global isr_43
1410061  .global isr_44
1410062  .global isr_45
1410063  .global isr_46
1410064  .global isr_47
1410065  .global isr_128
1410066 #-------------------------------------------------
1410067  .section .data
1410068 #-------------------------------------------------
1410069 proc_syscallnr:              .int   0x00000000
1410070 proc_msg_offset:            .int   0x00000000
1410071 proc_msg_size:              .int   0x00000000
1410072 proc_instruction_pointer:   .int   0x00000000
1410073 proc_back_address:          .int   0x00000000
1410074 _ksp:                       .int   0x00000000
1410075 syscall_working:            .int   0x00000000
1410076 _clock_kernel:
1410077 kticks_lo:                  .int   0x00000000
1410078 kticks_hi:                  .int   0x00000000
1410079 _clock_time:
1410080 tticks_lo:                  .int   0x00000000
1410081 tticks_hi:                  .int   0x00000000
1410082 #-------------------------------------------------
1410083  .section .text
1410084 #-------------------------------------------------
1410085 #
```



```
1410086 # Inside the stack there is already, placed by the CPU:
1410087 #   [omissis]
1410088 #   push %eflags
1410089 #   push %cs
1410090 #   push %eip
1410091 #
1410092 #-------------------------------------------------
1410093 isr_0:          # «division by zero exception»
1410094     cli
1410095     push $0     # Null error code.
1410096     push $0     # Exception number.
1410097     jmp exception_unrecoverable
1410098 #-------------------------------------------------
1410099 isr_1:          # «debug exception»
1410100     cli
1410101     push $0     # Null error code.
1410102     push $1     # Exception number.
1410103     jmp exception_unrecoverable
1410104 #-------------------------------------------------
1410105 isr_2:          # «non maskable interrupt exception»
1410106     cli
1410107     push $0     # Null error code.
1410108     push $2     # Exception number.
1410109     jmp exception_unrecoverable
1410110 #-------------------------------------------------
1410111 isr_3:          # «breakpoint exception»
1410112     cli
1410113     push $0     # Null error code.
1410114     push $3     # Exception number.
1410115     jmp exception_unrecoverable
1410116 #-------------------------------------------------
1410117 isr_4:          # «into detected overflow exception»
1410118     cli
1410119     push $0     # Null error code.
1410120     push $4     # Exception number.
1410121     jmp exception_unrecoverable
1410122 #-------------------------------------------------
1410123 isr_5:          # «out of bounds exception»
1410124     cli
1410125     push $0     # Null error code.
1410126     push $5     # Exception number.
1410127     jmp exception_unrecoverable
1410128 #-------------------------------------------------
1410129 isr_6:          # «invalid opcode exception»
1410130     cli
1410131     push $0     # Null error code.
1410132     push $6     # Exception number.
1410133     jmp exception_unrecoverable
1410134 #-------------------------------------------------
1410135 isr_7:          # «no coprocessor exception»
1410136     cli
1410137     push $0     # Null error code.
1410138     push $7     # Exception number.
1410139     jmp exception_unrecoverable
1410140 #-------------------------------------------------
1410141 isr_8:          # «double fault exception»
1410142     cli
1410143     #           # Error code already present.
1410144     push $8     # Exception number.
1410145     jmp exception_unrecoverable
1410146 #-------------------------------------------------
1410147 isr_9:          # «coprocessor segment overrun
1410148                 # exception»
1410149     cli
1410150     push $0     # Null error code.
1410151     push $9     # Exception number.
1410152     jmp exception_unrecoverable
1410153 #-------------------------------------------------
1410154 isr_10:         # «bad TSS exception»
1410155     cli
1410156     #           # Error code already present.
1410157     push $10    # Exception number.
1410158     jmp exception_unrecoverable
1410159 #-------------------------------------------------
1410160 isr_11:         # «segment not present exception»
1410161     cli
1410162     #           # Error code already present.
1410163     push $11    # Exception number.
1410164     jmp exception_unrecoverable
1410165 #-------------------------------------------------
1410166 isr_12:         # «stack fault exception»
1410167     cli
1410168     #           # Error code already present.
1410169     push $12    # Exception number.
1410170     jmp exception_unrecoverable
1410171 #-------------------------------------------------
1410172 isr_13:         # «general protection fault exception»
```

```
1410173     cli
1410174     #           # Error code already present.
1410175     push $13    # Exception number.
1410176     jmp exception_unrecoverable
1410177 #---------------------------------------------------
1410178 isr_14:         # «page fault exception»
1410179     cli
1410180     #           # Error code already present.
1410181     push $14    # Exception number.
1410182     jmp exception_unrecoverable
1410183 #---------------------------------------------------
1410184 isr_15:         # «unknown interrupt exception»
1410185     cli
1410186     push $0     # Null error code.
1410187     push $15    # Exception number.
1410188     jmp exception_unrecoverable
1410189 #---------------------------------------------------
1410190 isr_16:         # «coprocessor fault exception»
1410191     cli
1410192     push $0     # Null error code.
1410193     push $16    # Exception number.
1410194     jmp exception_unrecoverable
1410195 #---------------------------------------------------
1410196 isr_17:         # «alignment check exception»
1410197     cli
1410198     push $0     # Null error code.
1410199     push $17    # Exception number.
1410200     jmp exception_unrecoverable
1410201 #---------------------------------------------------
1410202 isr_18:         # «machine check exception»
1410203     cli
1410204     push $0     # Null error code.
1410205     push $18    # Exception number.
1410206     jmp exception_unrecoverable
1410207 #---------------------------------------------------
1410208 isr_19:         # «reserved exception»
1410209     cli
1410210     push $0     # Null error code.
1410211     push $19    # Exception number.
1410212     jmp exception_unrecoverable
1410213 #---------------------------------------------------
1410214 isr_20:         # «reserved exception»
1410215     cli
1410216     push $0     # Null error code.
1410217     push $20    # Exception number.
1410218     jmp exception_unrecoverable
1410219 #---------------------------------------------------
1410220 isr_21:         # «reserved exception»
1410221     cli
1410222     push $0     # Null error code.
1410223     push $21    # Exception number.
1410224     jmp exception_unrecoverable
1410225 #---------------------------------------------------
1410226 isr_22:         # «reserved exception»
1410227     cli
1410228     push $0     # Null error code.
1410229     push $22    # Exception number.
1410230     jmp exception_unrecoverable
1410231 #---------------------------------------------------
1410232 isr_23:         # «reserved exception»
1410233     cli
1410234     push $0     # Null error code.
1410235     push $23    # Exception number.
1410236     jmp exception_unrecoverable
1410237 #---------------------------------------------------
1410238 isr_24:         # «reserved exception»
1410239     cli
1410240     push $0     # Null error code.
1410241     push $24    # Exception number.
1410242     jmp exception_unrecoverable
1410243 #---------------------------------------------------
1410244 isr_25:         # «reserved exception»
1410245     cli
1410246     push $0     # Null error code.
1410247     push $25    # Exception number.
1410248     jmp exception_unrecoverable
1410249 #---------------------------------------------------
1410250 isr_26:         # «reserved exception»
1410251     cli
1410252     push $0     # Null error code.
1410253     push $26    # Exception number.
1410254     jmp exception_unrecoverable
1410255 #---------------------------------------------------
1410256 isr_27:         # «reserved exception»
1410257     cli
1410258     push $0     # Null error code.
1410259     push $27    # Exception number.
```



```
1410260     jmp exception_unrecoverable
1410261 #---------------------------------------------------
1410262 isr_28:         # «reserved exception»
1410263     cli
1410264     push $0     # Null error code.
1410265     push $28    # Exception number.
1410266     jmp exception_unrecoverable
1410267 #---------------------------------------------------
1410268 isr_29:         # «reserved exception»
1410269     cli
1410270     push $0     # Null error code.
1410271     push $29    # Exception number.
1410272     jmp exception_unrecoverable
1410273 #---------------------------------------------------
1410274 isr_30:         # «reserved exception»
1410275     cli
1410276     push $0     # Null error code.
1410277     push $30    # Exception number.
1410278     jmp exception_unrecoverable
1410279 #---------------------------------------------------
1410280 isr_31:         # «reserved exception»
1410281     cli
1410282     push $0     # Null error code.
1410283     push $31    # Exception number.
1410284     jmp exception_unrecoverable
1410285 #---------------------------------------------------
1410286 isr_32:         # IRQ 0: «timer»
1410287     cli
1410288     jmp irq_timer
1410289 #---------------------------------------------------
1410290 isr_33:         # IRQ 1: tastiera
1410291     cli
1410292     jmp irq_keyboard
1410293 #---------------------------------------------------
1410294 isr_34:         # IRQ 2: it is fired for IRQ 8 to 15.
1410295     cli
1410296     #
1410297     # IRQ 2 must be ON inside the file
1410298     # 'kernel/proc/proc_init.c', so
1410299     # that it is guaranteed that the PIC 1 is reset
1410300     # here.
1410301     #
1410302     call isr_irq_clear_pic1
1410303     #
1410304     # For IRQ 2 there is nothing else to do, because it
1410305     # is a link to the PIC 2 (IRQ 8 to 15).
1410306     #
1410307     iret
1410308 #---------------------------------------------------
1410309 isr_35:         # IRQ 3
1410310     cli
1410311     jmp irq_pic1
1410312 #---------------------------------------------------
1410313 isr_36:         # IRQ 4
1410314     cli
1410315     jmp irq_pic1
1410316 #---------------------------------------------------
1410317 isr_37:         # IRQ 5
1410318     cli
1410319     jmp irq_pic1
1410320 #---------------------------------------------------
1410321 isr_38:         # IRQ 6: floppy disk drive
1410322     cli
1410323     jmp irq_pic1
1410324 #---------------------------------------------------
1410325 isr_39:         # IRQ 7: LPT 1
1410326     cli
1410327     jmp irq_pic1
1410328 #---------------------------------------------------
1410329 isr_40:         # IRQ 8: «real time clock (RTC)»
1410330     cli
1410331     jmp irq_pic2
1410332 #---------------------------------------------------
1410333 isr_41:         # IRQ 9
1410334     cli
1410335     jmp irq_pic2
1410336 #---------------------------------------------------
1410337 isr_42:         # IRQ 10
1410338     cli
1410339     jmp irq_pic2
1410340 #---------------------------------------------------
1410341 isr_43:         # IRQ 11
1410342     cli
1410343     jmp irq_pic2
1410344 #---------------------------------------------------
1410345 isr_44:         # IRQ 12: mouse PS/2
1410346     cli
```

```
    jmp irq_pic2
#-------------------------------------------------------
isr_45:         # IRQ 13: math coprocessor
    cli
    jmp irq_pic2
#-------------------------------------------------------
isr_46:         # IRQ 14: primary IDE channel
    cli
    jmp irq_pic2
#-------------------------------------------------------
isr_47:         # IRQ 15: secondary IDE channel
    cli
    jmp irq_pic2
#-------------------------------------------------------
#
# Unrecoverable exceptions.
#
exception_unrecoverable:
    #
    # Previous pushes:
    #   [omissis]
    #   push %eflags
    #   push %cs
    #   push %eip
    #
    #   push $<error_number>
    #   push $<idt_item_number>
    #
    pushl %gs
    pushl %fs
    pushl %es
    pushl %ds
    pushl %edi
    pushl %esi
    pushl %ebp
    pushl %ebx
    pushl %edx
    pushl %ecx
    pushl %eax
    #
    call isr_exception_unrecoverable
    #
    popl %eax
    popl %ecx
    popl %edx
    popl %ebx
    popl %ebp
    popl %esi
    popl %edi
    popl %ds
    popl %es
    popl %fs
    popl %gs
    #
    # Remove the IDT item number and the error code.
    #
    add $4, %esp
    add $4, %esp
    #
    # Return from interrupt.
    #
    iret
#-------------------------------------------------------
#
# Unspecified IRQ. Currently unused.
#
irq:
    #
    # Previous pushes:
    #   [omissis]
    #   push %eflags
    #   push %cs
    #   push %eip
    #
    #   push $0
    #   push $<idt_item_number>
    #
    call isr_irq_clear
    #
    # Remove the IDT item number and the null error
    # code.
    #
    add $4, %esp
    add $4, %esp
    #
    # Return from interrupt.
    #
```



```
    iret
#-------------------------------------------------------
#
# Keyboard IRQ.
#
irq_keyboard:
    #
    # Previous pushes:
    #   [omissis]
    #   push %eflags
    #   push %cs
    #   push %eip
    #
    pushl %gs
    pushl %fs
    pushl %es
    pushl %ds
    pushl %edi
    pushl %esi
    pushl %ebp
    pushl %ebx
    pushl %edx
    pushl %ecx
    pushl %eax
    #
    # Set the data segments to the kernel data segment,
    # so that the following variables can be accessed.
    #
    mov  $16, %ax # DS, ES, FS and GS.
    mov  %ax, %ds
    mov  %ax, %es
    mov  %ax, %fs
    mov  %ax, %gs
    #
    # Check if a system call is already working: if so,
    # just leave (go to L1).
    #
    cmpl $1, syscall_working
    je L1
    #
    # Call the keyboard handler.
    #
    call kbd_isr
    #
L1: # Restore original registers and return.
    #
    jmp irq_pic1_pop_iret
#-------------------------------------------------------
#
# Generic IRQ from PIC 1
#
irq_pic1:
    #
    # Previous pushes:
    #   [omissis]
    #   push %eflags
    #   push %cs
    #   push %eip
    #
    pushl %gs
    pushl %fs
    pushl %es
    pushl %ds
    pushl %edi
    pushl %esi
    pushl %ebp
    pushl %ebx
    pushl %edx
    pushl %ecx
    pushl %eax
    #
    # Set the data segments to the kernel data segment,
    # so that the following variables can be accessed.
    #
    mov  $16, %ax # DS, ES, FS and GS.
    mov  %ax, %ds
    mov  %ax, %es
    mov  %ax, %fs
    mov  %ax, %gs
    #
    # Check if a system call is already working: if so,
    # just leave (go to L2).
    #
    cmpl $1, syscall_working
    je L2
    #
    # If we are here, no system call is working and a
```

```
1410521     # user process was interrupted.
1410522     # Save process stack registers into kernel data
1410523     # segment.
1410524     #
1410525     mov %ss,  proc_stack_segment_selector
1410526     mov %esp, proc_stack_pointer
1410527     #
1410528     # Check if it is already in kernel mode: the kernel
1410529     # has PID 0.
1410530     # If so, just leave (go to L2).
1410531     #
1410532     mov proc_current, %edx       # Interrupted PID.
1410533     mov $0,           %eax       # Kernel PID.
1410534     cmp %eax, %edx
1410535     je L5
1410536     #
1410537     # If we are here, a user process was interrupted.
1410538     # Switch to the kernel stack (data segment
1410539     # descriptor).
1410540     #
1410541     mov $16, %ax
1410542     mov %ax, %ss
1410543     mov _ksp, %esp
1410544     #
1410545     # Call the scheduler.
1410546     #
1410547     call proc_scheduler
1410548     #
1410549     # Restore process stack registers from kernel data
1410550     # segment.
1410551     #
1410552     mov proc_stack_segment_selector, %ss
1410553     mov proc_stack_pointer, %esp
1410554     #
1410555 L5: # Restore from process stack and return.
1410556     #
1410557     jmp irq_pic1_pop_iret
1410558 #-------------------------------------------------------
1410559 #
1410560 # Generic IRQ from PIC 2
1410561 #
1410562 irq_pic2:
1410563     #
1410564     # Previous pushes:
1410565     #   [omissis]
1410566     #   push %eflags
1410567     #   push %cs
1410568     #   push %eip
1410569     #
1410570     pushl %gs
1410571     pushl %fs
1410572     pushl %es
1410573     pushl %ds
1410574     pushl %edi
1410575     pushl %esi
1410576     pushl %ebp
1410577     pushl %ebx
1410578     pushl %edx
1410579     pushl %ecx
1410580     pushl %eax
1410581     #
1410582     # Set the data segments to the kernel data segment,
1410583     # so that the following variables can be accessed.
1410584     #
1410585     mov  $16, %ax # DS, ES, FS and GS.
1410586     mov  %ax, %ds
1410587     mov  %ax, %es
1410588     mov  %ax, %fs
1410589     mov  %ax, %gs
1410590     #
1410591     # Check if a system call is already working: if so,
1410592     # just leave (go to L2).
1410593     #
1410594     cmpl $1, syscall_working
1410595     je L2
1410596     #
1410597     # If we are here, no system call is working and a
1410598     # user process was interrupted.
1410599     # Save process stack registers into kernel data
1410600     # segment.
1410601     #
1410602     mov %ss,  proc_stack_segment_selector
1410603     mov %esp, proc_stack_pointer
1410604     #
1410605     # Check if it is already in kernel mode: the kernel
1410606     # has PID 0.
1410607     # If so, just leave (go to L2).
```

```
1410608     #
1410609     mov proc_current, %edx       # Interrupted PID.
1410610     mov $0,           %eax       # Kernel PID.
1410611     cmp %eax, %edx
1410612     je L4
1410613     #
1410614     # If we are here, a user process was interrupted.
1410615     # Switch to the kernel stack (data segment
1410616     # descriptor).
1410617     #
1410618     mov $16, %ax
1410619     mov %ax, %ss
1410620     mov _ksp, %esp
1410621     #
1410622     # Call the scheduler.
1410623     #
1410624     call proc_scheduler
1410625     #
1410626     # Restore process stack registers from kernel data
1410627     # segment.
1410628     #
1410629     mov proc_stack_segment_selector, %ss
1410630     mov proc_stack_pointer, %esp
1410631     #
1410632 L4: # Restore from process stack and return.
1410633     #
1410634     jmp irq_pic2_pop_iret
1410635 #-------------------------------------------------------
1410636 #
1410637 # Timer IRQ.
1410638 #
1410639 irq_timer:
1410640     #
1410641     # Previous pushes:
1410642     #   [omissis]
1410643     #   push %eflags
1410644     #   push %cs
1410645     #   push %eip
1410646     #
1410647     pushl %gs
1410648     pushl %fs
1410649     pushl %es
1410650     pushl %ds
1410651     pushl %edi
1410652     pushl %esi
1410653     pushl %ebp
1410654     pushl %ebx
1410655     pushl %edx
1410656     pushl %ecx
1410657     pushl %eax
1410658     #
1410659     # Set the data segments to the kernel data segment,
1410660     # so that the following variables can be accessed.
1410661     #
1410662     mov  $16, %ax # DS, ES, FS and GS.
1410663     mov  %ax, %ds
1410664     mov  %ax, %es
1410665     mov  %ax, %fs
1410666     mov  %ax, %gs
1410667     #
1410668     # Increment time counters, to keep time.
1410669     #
1410670     add $1, kticks_lo    # Kernel ticks counter.
1410671     adc $0, kticks_hi    #
1410672     #
1410673     add $1, tticks_lo    # Clock ticks counter.
1410674     adc $0, tticks_hi    #
1410675     #
1410676     # Check if a system call is already working: if so,
1410677     # just leave (go to L2).
1410678     #
1410679     cmpl $1, syscall_working
1410680     je L2
1410681     #
1410682     # If we are here, no system call is working and a
1410683     # user process was interrupted.
1410684     # Save process stack registers into kernel data
1410685     # segment.
1410686     #
1410687     mov %ss,  proc_stack_segment_selector
1410688     mov %esp, proc_stack_pointer
1410689     #
1410690     # Check if it is already in kernel mode: the kernel
1410691     # has PID 0.
1410692     # If so, just leave (go to L2).
1410693     #
1410694     mov proc_current, %edx       # Interrupted PID.
```

```
1410695     mov $0,           %eax       # Kernel PID.
1410696     cmp %eax, %edx
1410697     je L2
1410698     #
1410699     # If we are here, a user process was interrupted.
1410700     # Switch to the kernel stack (data segment
1410701     # descriptor).
1410702     #
1410703     mov $16, %ax
1410704     mov %ax, %ss
1410705     mov _ksp, %esp
1410706     #
1410707     # Call the scheduler.
1410708     #
1410709     call proc_scheduler
1410710     #
1410711     # Restore process stack registers from kernel data
1410712     # segment.
1410713     #
1410714     mov proc_stack_segment_selector, %ss
1410715     mov proc_stack_pointer, %esp
1410716     #
1410717 L2: # Restore from process stack and return.
1410718     #
1410719     jmp irq_pic1_pop_iret
1410720 #------------------------------------------------------
1410721 irq_pic1_pop_iret:
1410722     #
1410723     # Restore from process stack.
1410724     #
1410725     popl %eax
1410726     popl %ecx
1410727     popl %edx
1410728     popl %ebx
1410729     popl %ebp
1410730     popl %esi
1410731     popl %edi
1410732     popl %ds
1410733     popl %es
1410734     popl %fs
1410735     popl %gs
1410736     #
1410737     # End of hardware interrupt to PIC 1.
1410738     #
1410739     call isr_irq_clear_pic1
1410740     #
1410741     # Return from interrupt.
1410742     #
1410743     iret
1410744 #------------------------------------------------------
1410745 irq_pic2_pop_iret:
1410746     #
1410747     # Restore from process stack.
1410748     #
1410749     popl %eax
1410750     popl %ecx
1410751     popl %edx
1410752     popl %ebx
1410753     popl %ebp
1410754     popl %esi
1410755     popl %edi
1410756     popl %ds
1410757     popl %es
1410758     popl %fs
1410759     popl %gs
1410760     #
1410761     # End of hardware interrupt to PIC 2 and PIC1.
1410762     #
1410763     call isr_irq_clear_pic2
1410764     #
1410765     # Return from interrupt.
1410766     #
1410767     iret
1410768 #------------------------------------------------------
1410769 #
1410770 # System call.
1410771 #
1410772 isr_128:
1410773     #
1410774     # Previous pushes:
1410775     #   [omissis]
1410776     #   push message_size
1410777     #   push &message_structure
1410778     #   push syscall_number
1410779     #   push back_address   # made by a call to sys()
1410780     #   push %eflags        # made by int $128 (0x80)
1410781     #   push %cs            # made by int $128 (0x80)
```



```
1410782     #   push %eip           # made by int $128 (0x80)
1410783     #
1410784     #------------------------------------------------
1410785     #
1410786     # Save into process stack:
1410787     #
1410788     pushl %gs
1410789     pushl %fs
1410790     pushl %es
1410791     pushl %ds
1410792     pushl %edi
1410793     pushl %esi
1410794     pushl %ebp
1410795     pushl %ebx
1410796     pushl %edx
1410797     pushl %ecx
1410798     pushl %eax
1410799     #
1410800     # Set the data segments to the kernel data segment.
1410801     #
1410802     mov  $16, %ax # DS, ES, FS and GS.
1410803     mov  %ax, %ds
1410804     mov  %ax, %es
1410805     mov  %ax, %fs
1410806     mov  %ax, %gs
1410807     #
1410808     # Tell that it is a system call.
1410809     #
1410810     movl $1, syscall_working
1410811     #
1410812     # Save process stack registers into kernel data
1410813     # segment.
1410814     #
1410815     mov %ss,  proc_stack_segment_selector
1410816     mov %esp, proc_stack_pointer
1410817     #
1410818     # Save some more data, from the system call.
1410819     #
1410820     .equ SYSCALL_NUMBER,      60
1410821     .equ MESSAGE_OFFSET,      64
1410822     .equ MESSAGE_SIZE,        68
1410823     #
1410824     mov %esp, %ebp
1410825     mov SYSCALL_NUMBER(%ebp), %eax
1410826     mov %eax, proc_syscallnr
1410827     mov MESSAGE_OFFSET(%ebp), %eax
1410828     mov %eax, proc_msg_offset
1410829     mov MESSAGE_SIZE(%ebp), %eax
1410830     mov %eax, proc_msg_size
1410831     #
1410832     # Check if it is already in kernel mode: the kernel
1410833     # has PID 0.
1410834     #
1410835     mov proc_current, %edx       # Interrupted PID.
1410836     mov $0,           %eax       # Kernel PID.
1410837     cmp %eax, %edx
1410838     jne L3
1410839     #
1410840     # It is already the kernel stack, so, the variable
1410841     # "_ksp" is aligned to current stack pointer.
1410842     # This way, the first syscall
1410843     # can work without having to set the "_ksp"
1410844     # variable to some reasonable value.
1410845     #
1410846     mov %esp, _ksp
1410847     #
1410848 L3: # Switch to the kernel stack (data segment
1410849     # descriptor).
1410850     #
1410851     mov $16, %ax
1410852     mov %ax, %ss
1410853     mov _ksp, %esp
1410854     #
1410855     # Call the external sysroutine handler.
1410856     #
1410857     push proc_msg_size
1410858     push proc_msg_offset
1410859     push proc_syscallnr
1410860     call sysroutine
1410861     add $4, %esp
1410862     add $4, %esp
1410863     add $4, %esp
1410864     #
1410865     # Restore process stack registers from kernel data
1410866     # segment.
1410867     #
1410868     mov proc_stack_segment_selector, %ss
```

```
1410869     mov proc_stack_pointer, %esp
1410870     #
1410871     # End of system call.
1410872     #
1410873     movl $0, syscall_working
1410874     #
1410875     # Restore from process stack.
1410876     #
1410877     popl %eax
1410878     popl %ecx
1410879     popl %edx
1410880     popl %ebx
1410881     popl %ebp
1410882     popl %esi
1410883     popl %edi
1410884     popl %ds
1410885     popl %es
1410886     popl %fs
1410887     popl %gs
1410888     #
1410889     # Return from interrupt.
1410890     #
1410891     iret
1410892 #---------------------------------------------------
```

94.6.22 kernel/ibm_i386/isr_exception_name.c

Si veda la sezione 93.7.

```
1420001 #include <kernel/ibm_i386.h>
1420002 //-----------------------------------------------------------
1420003 char *
1420004 isr_exception_name (int exception)
1420005 {
1420006   char *description[19] = { "division by zero",
1420007     "debug",
1420008     "non maskable interrupt",
1420009     "breakpoint",
1420010     "into detected overflow",
1420011     "out of bounds",
1420012     "invalid opcode",
1420013     "no coprocessor",
1420014     "double fault",
1420015     "coprocessor segmento overrun",
1420016     "bad TSS",
1420017     "segment not present",
1420018     "stack fault",
1420019     "general protection fault",
1420020     "page fault",
1420021     "unknown interrupt",
1420022     "coprocessor fault",
1420023     "alignment check",
1420024     "machine check"
1420025   };
1420026   //
1420027   if (exception >= 0 && exception <= 18)
1420028     {
1420029       return description[exception];
1420030     }
1420031   else
1420032     {
1420033       return "unknown";
1420034     }
1420035 }
```

94.6.23 kernel/ibm_i386/isr_exception_unrecoverable.c

Si veda la sezione 93.7.

```
1430001 #include <kernel/ibm_i386.h>
1430002 #include <kernel/lib_k.h>
1430003 #include <sys/types.h>
1430004 //-----------------------------------------------------------
1430005 void
1430006 isr_exception_unrecoverable (uint32_t eax,
1430007                              uint32_t ecx,
1430008                              uint32_t edx,
1430009                              uint32_t ebx,
1430010                              uint32_t ebp,
1430011                              uint32_t esi,
1430012                              uint32_t edi, uint32_t ds,
1430013                              uint32_t es, uint32_t fs,
1430014                              uint32_t gs,
1430015                              uint32_t interrupt,
1430016                              uint32_t error,
1430017                              uint32_t eip, uint32_t cs,
1430018                              uint32_t eflags)
```



```
1430019 {
1430020   pid_t pid = cs / 8;
1430021   //
1430022   k_printf
1430023     ("[%s] ERROR: pid: %i exception %i: \"%s\"\n",
1430024      __func__, pid, interrupt,
1430025      isr_exception_name (interrupt));
1430026   //
1430027   // Exit the unrecoverable application.
1430028   //
1430029   k_exit ();
1430030 }
```

94.6.24 kernel/ibm_i386/isr_irq_clear.c

Si veda la sezione 93.7.

```
1440001 #include <kernel/ibm_i386.h>
1440002 //-----------------------------------------------------------
1440003 void
1440004 isr_irq_clear (uint32_t idtn)
1440005 {
1440006   int irq = idtn - 32;
1440007   //
1440008   // Must tell the PIC (programmable interrupt
1440009   // controller).
1440010   //
1440011   // If the IRQ number is between 8 and 15, send
1440012   // message «EOI»
1440013   // (End of IRQ) to PIC 2.
1440014   //
1440015   if (irq >= 8)
1440016     {
1440017       out_8 (0xA0, 0x20);
1440018     }
1440019   //
1440020   // Then send message «EOI» to PIC 1.
1440021   //
1440022   out_8 (0x20, 0x20);
1440023 }
```

94.6.25 kernel/ibm_i386/isr_irq_clear_pic1.c

Si veda la sezione 93.7.

```
1450001 #include <kernel/ibm_i386.h>
1450002 //-----------------------------------------------------------
1450003 void
1450004 isr_irq_clear_pic1 (void)
1450005 {
1450006   //
1450007   // Send message «EOI» to PIC 1.
1450008   //
1450009   out_8 ((uint32_t) 0x20, (uint32_t) 0x20);
1450010 }
```

94.6.26 kernel/ibm_i386/isr_irq_clear_pic2.c

Si veda la sezione 93.7.

```
1460001 #include <kernel/ibm_i386.h>
1460002 //-----------------------------------------------------------
1460003 void
1460004 isr_irq_clear_pic2 (void)
1460005 {
1460006   //
1460007   // Send message «EOI» (End of IRQ) to PIC 2.
1460008   // It must be sent after a IRQ number between 8 and
1460009   // 15, but after
1460010   // that, remember that also the PIC 1 must receive
1460011   // an «EOI» message
1460012   // (maybe with the help of 'isr_irq_clear_pic1()'
1460013   // function).
1460014   //
1460015   out_8 ((uint32_t) 0xA0, (uint32_t) 0x20);
1460016 }
```

94.6.27 kernel/ibm_i386/sti.s

Si veda la sezione 93.7.

```
1470001 .global sti
1470002 #---------------------------------------------------
1470003 .text
1470004 #---------------------------------------------------
1470005 # Set interrupt flag.
```

```
1470006 #-----------------------------------------------------
1470007 .align 4
1470008 sti:
1470009     sti
1470010     ret
```

94.7 os32: «kernel/lib_k.h»

Si veda la sezione 93.11.

```
1480001 #ifndef _KERNEL_LIB_K_H
1480002 #define _KERNEL_LIB_K_H        1
1480003 //-------------------------------------------------------
1480004 #include <restrict.h>
1480005 #include <size_t.h>
1480006 #include <stdarg.h>
1480007 #include <stdint.h>
1480008 #include <time.h>
1480009 #include <unistd.h>
1480010 #include <kernel/lib_s.h>
1480011 //-------------------------------------------------------
1480012 void k_exit (void);
1480013 unsigned int k_sleep (unsigned int seconds);
1480014 int k_usleep (useconds_t usec);
1480015 char *k_gets (char *s);
1480016 void k_perror (const char *s);
1480017 int k_printf (const char *restrict format, ...);
1480018 int k_stime (time_t * timer);
1480019 int k_vprintf (const char *restrict format, va_list arg);
1480020 int k_vsprintf (char *restrict string,
1480021                 const char *restrict format, va_list arg);
1480022 //clock_t      k_clock    (void);
1480023 #define      k_clock()  (s_clock ((uid_t) 0))
1480024 #define      k_time(t)  (s_time ((pid_t) 0, t))
1480025 //-------------------------------------------------------
1480026 #endif
```


94.7.1 kernel/lib_k/k_exit.s

Si veda la sezione 93.11.

```
1490001 .global k_exit
1490002 #-----------------------------------------------------
1490003 .text
1490004 #-----------------------------------------------------
1490005 .align 4
1490006 k_exit:
1490007 halt:
1490008     hlt
1490009     jmp halt
```

94.7.2 kernel/lib_k/k_gets.c

Si veda la sezione 93.11.

```
1500001 #include <kernel/lib_k.h>
1500002 #include <kernel/driver/kbd.h>
1500003 //-------------------------------------------------------
1500004 char *
1500005 k_gets (char *s)
1500006 {
1500007   int i;
1500008   //
1500009   // Legge kbd.char.
1500010   //
1500011   for (i = 0; i < 256; i++)
1500012     {
1500013       while (kbd.key == 0)
1500014         {
1500015           //
1500016           // Attende un carattere.
1500017           //
1500018           ;
1500019         }
1500020       s[i] = kbd.key;
1500021       kbd.key = 0;
1500022       if (s[i] == '\n')
1500023         {
1500024           s[i] = 0;
1500025           break;
1500026         }
1500027     }
1500028   return s;
1500029 }
```


94.7.3 kernel/lib_k/k_perror.c

Si veda la sezione 93.11.

```
1510001 #include <kernel/lib_k.h>
1510002 #include <errno.h>
1510003 //-------------------------------------------------------
1510004 void
1510005 k_perror (const char *s)
1510006 {
1510007   //
1510008   // If errno is zero, there is nothing to show.
1510009   //
1510010   if (errno == 0)
1510011     {
1510012       return;
1510013     }
1510014   //
1510015   // Show the string if there is one.
1510016   //
1510017   if (s != NULL && strlen (s) > 0)
1510018     {
1510019       k_printf ("%s: ", s);
1510020     }
1510021   //
1510022   // Show the translated error.
1510023   //
1510024   if (errfn[0] != 0 && errln != 0)
1510025     {
1510026       k_printf ("[%s:%u:%i] %s\n",
1510027                 errfn, errln, errno, strerror (errno));
1510028     }
1510029   else
1510030     {
1510031       k_printf ("[%i] %s\n", errno, strerror (errno));
1510032     }
1510033 }
```

94.7.4 kernel/lib_k/k_printf.c

Si veda la sezione 93.11.

```
1520001 #include <stdarg.h>
1520002 #include <kernel/lib_k.h>
1520003 //-------------------------------------------------------
1520004 int
1520005 k_printf (const char *restrict format, ...)
1520006 {
1520007   va_list ap;
1520008   va_start (ap, format);
1520009   return k_vprintf (format, ap);
1520010 }
```

94.7.5 kernel/lib_k/k_sleep.c

Si veda la sezione 93.11.

```
1530001 #include <kernel/lib_k.h>
1530002 #include <kernel/lib_s.h>
1530003 #include <kernel/proc.h>
1530004 #include <time.h>
1530005 //-------------------------------------------------------
1530006 unsigned int
1530007 k_sleep (unsigned int seconds)
1530008 {
1530009   clock_t time_start;
1530010   clock_t time_now;
1530011   clock_t time_elapsed;
1530012   clock_t time = seconds * CLOCKS_PER_SEC;
1530013   unsigned long long int loops;
1530014   //
1530015   // Calculate how many times the following loop
```

```
1530016   // should
1530017   // be run to get the requested time.
1530018   //
1530019   loops = proc_loops_per_clock * time;
1530020   //
1530021   // Do a wasting time loop.
1530022   //
1530023   time_elapsed = 0;
1530024   time_start = s_clock ((pid_t) 0);
1530025   //
1530026   // If the function 's_clock()' can help, it will
1530027   // exit even if the loop is become slow, but the
1530028   // time was correctly accounted and is elapsed.
1530029   //
1530030   for (; time_elapsed < time && loops > 0; loops--)
1530031     {
1530032       time_now = s_clock ((pid_t) 0);
1530033       time_elapsed = time_now - time_start;
1530034     }
1530035   //
1530036   // The sleep is always complete.
1530037   //
1530038   return (0);
1530039 }
```

94.7.6 kernel/lib_k/k_stime.c

«

Si veda la sezione 93.11.

```
1540001 #include <kernel/lib_k.h>
1540002 //-------------------------------------------------------
1540003 extern clock_t _clock_time;     // uint64_t
1540004 //-------------------------------------------------------
1540005 int
1540006 k_stime (time_t * timer)
1540007 {
1540008   _clock_time = (*timer * CLOCKS_PER_SEC);
1540009   return (0);
1540010 }
```

94.7.7 kernel/lib_k/k_usleep.c

«

Si veda la sezione 93.11.

```
1550001 #include <kernel/lib_k.h>
1550002 #include <kernel/lib_s.h>
1550003 #include <kernel/proc.h>
1550004 #include <time.h>
1550005 #include <unistd.h>
1550006 //-------------------------------------------------------
1550007 int
1550008 k_usleep (useconds_t usec)
1550009 {
1550010   clock_t time_start;
1550011   clock_t time_now;
1550012   clock_t time_elapsed;
1550013   clock_t time;
1550014   unsigned long long int loops;
1550015   //
1550016   // Calculate time, in terms of internal clocks
1550017   //
1550018   if (usec < 10000000)
1550019     {
1550020       time = (usec * CLOCKS_PER_SEC) / 1000000;
1550021     }
1550022   else
1550023     {
1550024       time = (usec / 1000000) * CLOCKS_PER_SEC;
1550025     }
1550026   //
1550027   // Fix time: if it is zero, it means that it was
1550028   // requested a sleep
1550029   // shorter than the internal clock timer impulse.
1550030   // So, if it is zero,
1550031   // correct to at least a one.
1550032   //
1550033   if (time == 0 && usec != 0)
1550034     time = 1;
1550035   //
1550036   // Calculate how many times the following loop
1550037   // should
1550038   // be run to get the requested time.
1550039   //
1550040   loops = proc_loops_per_clock * time;
1550041   //
1550042   // Do a wasting time loop.
1550043   //
```



```
1550044   time_elapsed = 0;
1550045   time_start = s_clock ((pid_t) 0);
1550046   //
1550047   // If the function 's_clock()' can help, it will
1550048   // exit even if the loop is become slow, but the
1550049   // time was correctly accounted and is elapsed.
1550050   //
1550051   for (; time_elapsed < time && loops > 0; loops--)
1550052     {
1550053       time_now = s_clock ((pid_t) 0);
1550054       time_elapsed = time_now - time_start;
1550055     }
1550056   //
1550057   // The sleep is always complete.
1550058   //
1550059   return (0);
1550060 }
```

94.7.8 kernel/lib_k/k_vprintf.c

«

Si veda la sezione 93.11.

```
1560001 #include <kernel/lib_k.h>
1560002 #include <stdio.h>
1560003 #include <kernel/dev.h>
1560004 //-------------------------------------------------------
1560005 int
1560006 k_vprintf (const char *restrict format, va_list arg)
1560007 {
1560008   size_t size = BUFSIZ;
1560009   char string[BUFSIZ];
1560010   int status;
1560011   //
1560012   // At the moment, set 'string' to be a null string.
1560013   //
1560014   string[0] = 0;
1560015   //
1560016   // Get the string, that must be limited to 'size'
1560017   // bytes.
1560018   //
1560019   status = vsnprintf (string, size, format, arg);
1560020   //
1560021   //
1560022   //
1560023   if (status < 0)
1560024     {
1560025       //
1560026       // Variable 'errno' is not updated.
1560027       //
1560028       return (status);
1560029     }
1560030   //
1560031   // Get size.
1560032   //
1560033   size = status;
1560034   //
1560035   // Write to the first console.
1560036   //
1560037   dev_io ((pid_t) 0, DEV_CONSOLE0, DEV_WRITE,
1560038           (off_t) 0, string, size, NULL);
1560039   //
1560040   // Return the same value obtained from 'vsnprintf()'
1560041   //
1560042   return status;
1560043 }
```

94.7.9 kernel/lib_k/k_vsprintf.c

«

Si veda la sezione 93.11.

```
1570001 #include <stdarg.h>
1570002 #include <kernel/lib_k.h>
1570003 #include <stdio.h>
1570004 //-------------------------------------------------------
1570005 int
1570006 k_vsprintf (char *restrict string,
1570007              const char *restrict format, va_list arg)
1570008 {
1570009   int status;
1570010   status = vsnprintf (string, BUFSIZ, format, arg);
1570011   return status;
1570012 }
```

94.8 os32: «kernel/lib_s.h»

«

Si veda la sezione 93.12.

```
1580001 #ifndef _KERNEL_LIB_S_H
1580002 #define _KERNEL_LIB_S_H  1
1580003 //-------------------------------------------------------------
1580004 #include <sys/types.h>
1580005 #include <sys/stat.h>
1580006 #include <kernel/fs.h>
1580007 #include <sys/os32.h>
1580008 #include <stddef.h>
1580009 #include <stdint.h>
1580010 #include <time.h>
1580011 #include <termios.h>
1580012 #include <setjmp.h>
1580013 //-------------------------------------------------------------
1580014 void s__exit (pid_t pid, int status);
1580015 int s_brk (pid_t pid, void *address);
1580016 void *s_sbrk (pid_t pid, intptr_t increment);
1580017 pid_t s_fork (pid_t ppid);
1580018 int s_kill (pid_t pid_killer, pid_t pid_target, int sig);
1580019 void s_longjmp (pid_t pid, jmp_buf env, int val);
1580020 int s_seteuid (pid_t pid, uid_t euid);
1580021 int s_setjmp (pid_t pid, jmp_buf env);
1580022 int s_setuid (pid_t pid, uid_t uid);
1580023 int s_setegid (pid_t pid, gid_t egid);
1580024 int s_setgid (pid_t pid, gid_t gid);
1580025 pid_t s_wait (pid_t pid, int *status);
1580026 sighandler_t s_signal (pid_t pid, int sig,
1580027                        sighandler_t handler,
1580028                        uintptr_t wrapper);
1580029 //-------------------------------------------------------------
1580030 int s_chdir (pid_t pid, const char *path);
1580031 int s_chmod (pid_t pid, const char *path, mode_t mode);
1580032 int s_chown (pid_t pid, const char *path, uid_t uid,
1580033              gid_t gid);
1580034 int s_link (pid_t pid, const char *path_old,
1580035             const char *path_new);
1580036 int s_mkdir (pid_t pid, const char *path, mode_t mode);
1580037 int s_mknod (pid_t pid, const char *path, mode_t mode,
1580038              dev_t device);
1580039 int s_open (pid_t pid, const char *path, int oflags,
1580040             mode_t mode);
1580041 int s_stat (pid_t pid, const char *path,
1580042             struct stat *buffer);
1580043 int s_unlink (pid_t pid, const char *path);
1580044 //
1580045 int s_pipe (pid_t pid, int pipefd[2]);
1580046 //
1580047 int s_close (pid_t pid, int fdn);
1580048 int s_dup (pid_t pid, int fdn_old);
1580049 int s_dup2 (pid_t pid, int fdn_old, int fdn_new);
1580050 int s_fchmod (pid_t pid, int fdn, mode_t mode);
1580051 int s_fchown (pid_t pid, int fdn, uid_t uid, gid_t gid);
1580052 int s_fcntl (pid_t pid, int fdn, int cmd, int arg);
1580053 int s_fstat (pid_t pid, int fdn, struct stat *buffer);
1580054 off_t s_lseek (pid_t pid, int fdn, off_t offset,
1580055                int whence);
1580056 ssize_t s_read (pid_t pid, int fdn, void *buffer,
1580057                 size_t count);
1580058 ssize_t s_write (pid_t pid, int fdn,
1580059                  const void *buffer, size_t count);
1580060 //
1580061 int s_mount (pid_t pid, const char *path_dev,
1580062              const char *path_mnt, int options);
1580063 int s_umount (pid_t pid, const char *path_mnt);
1580064 //-------------------------------------------------------------
1580065 int s_accept (pid_t pid, int sfdn,
1580066               struct sockaddr *addr, socklen_t * addrlen);
1580067 int s_bind (pid_t pid, int sfdn,
1580068             const struct sockaddr *addr, socklen_t addrlen);
1580069 int s_connect (pid_t pid, int sfdn,
1580070               const struct sockaddr *addr,
1580071               socklen_t addrlen);
1580072 int s_listen (pid_t pid, int sfdn, int backlog);
1580073 int s_socket (pid_t pid, int family, int type,
1580074               int protocol);
1580075 ssize_t s_send (pid_t pid, int sfdn,
1580076                 const void *buffer, size_t size, int flags);
1580077 ssize_t s_recv (pid_t pid, int sfdn, void *buffer,
1580078                 size_t size, int flags);
1580079 ssize_t s_recvfrom (pid_t pid, int sfdn, void *buffer,
1580080                     size_t length, int flags,
1580081                     struct sockaddr *addrfrom,
1580082                     socklen_t * addrlen);
1580083 //
1580084 int s_ipconfig (pid_t pid, int n, h_addr_t address, int m);
```



```
1580085 int s_routeadd (pid_t pid, h_addr_t destination, int m,
1580086                  h_addr_t router, int device);
1580087 int s_routedel (pid_t pid, h_addr_t destination, int m);
1580088 //-------------------------------------------------------------
1580089 int s_tcgetattr (pid_t pid, int fdn,
1580090                  struct termios *termios_p);
1580091 int s_tcsetattr (pid_t pid, int fdn, int action,
1580092                  struct termios *termios_p);
1580093 //-------------------------------------------------------------
1580094 clock_t s_clock (pid_t pid);    // [p]
1580095 time_t s_time (pid_t pid, time_t * timer);      // [p]
1580096 int s_stime (pid_t pid, time_t * timer);
1580097 //
1580098 // [p] The PID information, here, is not used. The
1580099 //     argument is required only for syntax coherence
1580100 //     with other functions, that do
1580101 //     system calls, inside the kernel.
1580102 //
1580103 //-------------------------------------------------------------
1580104 #endif
```

94.8.1 kernel/lib_s/s__exit.c

«

Si veda la sezione 87.2.

```
1590001 #include <errno.h>
1590002 #include <kernel/proc.h>
1590003 #include <kernel/lib_k.h>
1590004 #include <kernel/lib_s.h>
1590005 //-----------------------------------------------------------
1590006 void
1590007 s__exit (pid_t pid, int status)
1590008 {
1590009   pid_t child;
1590010   pid_t parent = proc_table[pid].ppid;
1590011   int proc_count;
1590012   pid_t extra;
1590013   int sigchld = 0;
1590014   int fdn;
1590015   tty_t *tty;
1590016   int closed;
1590017   fd_t *fd;
1590018   //
1590019   proc_table[pid].status = PROC_ZOMBIE;
1590020   proc_table[pid].ret = status;
1590021   proc_table[pid].sig_status = 0;
1590022   proc_table[pid].sig_ignore = 0;
1590023   //
1590024   // Close files.
1590025   //
1590026   for (fdn = 0; fdn < OPEN_MAX; fdn++)
1590027     {
1590028       closed = s_close (pid, fdn);
1590029       //
1590030       // Close might fail for work in progress.
1590031       //
1590032       if (closed < 0 && errno == EINPROGRESS)
1590033         {
1590034           //
1590035           // Should be a socket, but close badly,
1590036           // because we cannot wait.
1590037           //
1590038           fd = fd_reference (pid, &fdn);
1590039           if (fd->file->sock != NULL)
1590040             {
1590041               fd->file->sock->active = 0;
1590042               s_close (pid, fdn);
1590043             }
1590044         }
1590045     }
1590046   //
1590047   // Close current directory.
1590048   //
1590049   inode_put (proc_table[pid].inode_cwd);
1590050   //
1590051   // Close the controlling terminal, if it is a
1590052   // process leader with
1590053   // such a terminal.
1590054   //
1590055   if (proc_table[pid].pgrp == pid
1590056       && proc_table[pid].device_tty != 0)
1590057     {
1590058       tty = tty_reference (proc_table[pid].device_tty);
1590059       //
1590060       // Verify.
1590061       //
1590062       if (tty == NULL)
1590063         {
```



```
1590064           //
1590065           // Show a kernel message.
1590066           //
1590067           k_printf
1590068             ("kernel alert: cannot find the "
1590069              "terminal item "
1590070              "for device 0x%04x!\n",
1590071              (int) proc_table[pid].device_tty);
1590072         }
1590073       else if (tty->pgrp != pid)
1590074         {
1590075           //
1590076           // Show a kernel message.
1590077           //
1590078           k_printf
1590079             ("kernel alert: terminal "
1590080              "device 0x%04x should "
1590081              "be associated to the "
1590082              "process group %i, but it "
1590083              "is instead related to "
1590084              "process group %i!\n",
1590085              (int) proc_table[pid].device_tty,
1590086              (int) pid, (int) tty->pgrp);
1590087         }
1590088       else
1590089         {
1590090           tty->pgrp = 0;
1590091         }
1590092     }
1590093   //
1590094   // Data and text might share the same address space.
1590095   // If they are are on different places, then must
1590096   // verify if the text is not shared by other
1590097   // processes.
1590098   //
1590099   if (proc_table[pid].domain_data == 0)
1590100     {
1590101       //
1590102       // Text and data are together.
1590103       //
1590104       mb_free (proc_table[pid].address_text,
1590105                (proc_table[pid].domain_text
1590106                 + proc_table[pid].extra_data));
1590107     }
1590108   else
1590109     {
1590110       //
1590111       // Data is separate and is to be removed alone.
1590112       //
1590113       mb_free (proc_table[pid].address_data,
1590114                (proc_table[pid].domain_data
1590115                 + proc_table[pid].extra_data));
1590116       //
1590117       // Now must verify if no other process uses the
1590118       // same text
1590119       // memory.
1590120       //
1590121       for (proc_count = 0, extra = 0;
1590122            extra < PROCESS_MAX; extra++)
1590123         {
1590124           if (proc_table[extra].status == PROC_EMPTY ||
1590125               proc_table[extra].status == PROC_ZOMBIE)
1590126             {
1590127               continue;
1590128             }
1590129           if (proc_table[pid].address_text
1590130               == proc_table[extra].address_text)
1590131             {
1590132               proc_count++;
1590133             }
1590134         }
1590135       if (proc_count == 0)
1590136         {
1590137           //
1590138           // The code segment can be released, because
1590139           // no other process, except the current one
1590140           // (to be closed), is using it.
1590141           //
1590142           mb_free (proc_table[pid].address_text,
1590143                    proc_table[pid].domain_text);
1590144         }
1590145     }
1590146   //
1590147   // Abandon children to 'init' ((pid_t) 1).
1590148   //
1590149   for (child = 1; child < PROCESS_MAX; child++)
1590150     {
```

```
1590151         if (proc_table[child].status != PROC_EMPTY
1590152             && proc_table[child].ppid == pid)
1590153           {
1590154             proc_table[child].ppid = 1;   // Son of
1590155             // 'init'.
1590156             if (proc_table[child].status == PROC_ZOMBIE)
1590157               {
1590158                 sigchld = 1;      // Must send a SIGCHLD
1590159                 // to 'init'.
1590160               }
1590161           }
1590162       }
1590163     //
1590164     // SIGCHLD to 'init'.
1590165     //
1590166     if (sigchld
1590167         && pid != 1
1590168         && proc_table[1].status != PROC_EMPTY
1590169         && proc_table[1].status != PROC_ZOMBIE)
1590170       {
1590171         proc_sig_on ((pid_t) 1, SIGCHLD);
1590172       }
1590173     //
1590174     // Announce to the parent the death of its child.
1590175     //
1590176     if (pid != parent
1590177         && proc_table[parent].status != PROC_EMPTY)
1590178       {
1590179         proc_sig_on (parent, SIGCHLD);
1590180       }
1590181   }
```

94.8.2 kernel/lib_s/s_accept.c

«

Si veda la sezione 87.3.

```
1600001 #include <kernel/proc.h>
1600002 #include <kernel/lib_s.h>
1600003 #include <kernel/lib_k.h>
1600004 #include <errno.h>
1600005 #include <fcntl.h>
1600006 #include <sys/socket.h>
1600007 #include <arpa/inet.h>
1600008 //-----------------------------------------------------------
1600009 int
1600010 s_accept (pid_t pid, int sfdn, struct sockaddr *addr,
1600011           socklen_t * addrlen)
1600012 {
1600013   fd_t *sfd;
1600014   fd_t *sfd2;
1600015   int sfdn2;
1600016   struct sockaddr_in sa;
1600017   int q;
1600018   //
1600019   // Get file descriptor and verify that it is a
1600020   // socket.
1600021   //
1600022   sfd = fd_reference (pid, &sfdn);
1600023   if (sfd == NULL || sfd->file == NULL)
1600024     {
1600025       errset (EBADF);   // Bad file descriptor.
1600026       return (-1);
1600027     }
1600028   if (sfd->file->sock == NULL)
1600029     {
1600030       errset (ENOTSOCK);        // Not a socket.
1600031       return (-1);
1600032     }
1600033   //
1600034   // The socket must be a stream.
1600035   //
1600036   if (sfd->file->sock->type != SOCK_STREAM)
1600037     {
1600038       errset (EOPNOTSUPP);
1600039       return (-1);
1600040     }
1600041   //
1600042   // The socket must be a TCP stream: no other stream
1600043   // types are
1600044   // available.
1600045   //
1600046   if (sfd->file->sock->protocol != IPPROTO_TCP)
1600047     {
1600048       errset (EOPNOTSUPP);
1600049       return (-1);
1600050     }
1600051   //
```



```
1600052   // The socket itself must be listening.
1600053   //
1600054   if (sfd->file->sock->tcp.conn != TCP_LISTEN)
1600055     {
1600056       errset (EINVAL);
1600057       return (-1);
1600058     }
1600059   //
1600060   // The connections must be related to the same PID.
1600061   //
1600062   if (sfd->file->sock->tcp.listen_pid != pid)
1600063     {
1600064       k_printf
1600065         ("[%s] the connection pid %i is not the same "
1600066          "as the current pid %i!\n", __func__,
1600067          sfd->file->sock->tcp.listen_pid, pid);
1600068       errset (EUNKNOWN);
1600069       return (-1);
1600070     }
1600071   //
1600072   // Ok: find a connected socket from the queue.
1600073   //
1600074   for (q = 0; q < sfd->file->sock->tcp.listen_max; q++)
1600075     {
1600076       if (sfd->file->sock->tcp.listen_queue[q] != -1)
1600077         {
1600078           //
1600079           // Found.
1600080           //
1600081           break;
1600082         }
1600083     }
1600084   if (q >= sfd->file->sock->tcp.listen_max)
1600085     {
1600086       //
1600087       // At the moment, there is no new connection.
1600088       //
1600089       errset (EAGAIN);  // Try again.
1600090       return (-1);
1600091     }
1600092   //
1600093   // Descriptor found: check it.
1600094   //
1600095   sfdn2 = sfd->file->sock->tcp.listen_queue[q];
1600096   sfd2 = fd_reference (pid, &sfdn2);
1600097   if (sfd2 == NULL || sfd->file == NULL)
1600098     {
1600099       k_printf
1600100         ("[%s] the connected file descriptor %i "
1600101          "for process %i is not a file descriptor!",
1600102          __func__, sfdn2, pid);
1600103       errset (EBADF);   // Bad file descriptor.
1600104       return (-1);
1600105     }
1600106   if (sfd2->file->sock == NULL)
1600107     {
1600108       k_printf
1600109         ("[%s] the connected file descriptor %i "
1600110          "for process %i is not a socket!", __func__,
1600111          sfdn2, pid);
1600112       errset (ENOTSOCK);        // Not a socket.
1600113       return (-1);
1600114     }
1600115   //
1600116   // Ok.
1600117   //
1600118   if (addrlen != NULL && addr != NULL && *addrlen > 0)
1600119     {
1600120       sa.sin_family = AF_INET;
1600121       sa.sin_port = htons (sfd2->file->sock->rport);
1600122       sa.sin_addr.s_addr = htonl (sfd2->file->sock->raddr);
1600123       //
1600124       memcpy (addr, &sa,
1600125               min (sizeof (sa), (size_t) * addrlen));
1600126       *addrlen = sizeof (sa);
1600127     }
1600128   //
1600129   // Reset the queue element and return.
1600130   //
1600131   sfd->file->sock->tcp.listen_queue[q] = -1;
1600132   return (sfdn2);
1600133 }
```

94.8.3 kernel/lib_s/s_bind.c

«

Si veda la sezione 87.4.

```
1610001 #include <kernel/proc.h>
1610002 #include <kernel/lib_s.h>
1610003 #include <kernel/lib_k.h>
1610004 #include <errno.h>
1610005 #include <fcntl.h>
1610006 #include <sys/socket.h>
1610007 #include <arpa/inet.h>
1610008 //-----------------------------------------------------------
1610009 int
1610010 s_bind (pid_t pid, int sfdn,
1610011         const struct sockaddr *addr, socklen_t addrlen)
1610012 {
1610013   fd_t *sfd;
1610014   struct sockaddr_in *sin;
1610015   proc_t *ps = proc_reference (pid);
1610016   int i;
1610017   clock_t clock_time;
1610018   //
1610019   // Get file descriptor and verify that it is a
1610020   // socket.
1610021   //
1610022   sfd = fd_reference (pid, &sfdn);
1610023   if (sfd == NULL || sfd->file == NULL)
1610024     {
1610025       errset (EBADF);   // Bad file descriptor.
1610026       return (-1);
1610027     }
1610028   if (sfd->file->sock == NULL)
1610029     {
1610030       errset (ENOTSOCK);        // Not a socket.
1610031       return (-1);
1610032     }
1610033   //
1610034   // Verify to have a valid address pointer.
1610035   //
1610036   if (addr == NULL)
1610037     {
1610038       errset (EINVAL);
1610039       return (-1);
1610040     }
1610041   //
1610042   // Check minimal address size.
1610043   //
1610044   if (addrlen < sizeof (struct sockaddr))
1610045     {
1610046       errset (EINVAL);
1610047       return (-1);
1610048     }
1610049   //
1610050   //
1610051   //
1610052   if (addr->sa_family == AF_INET)
1610053     {
1610054       sin = (struct sockaddr_in *) addr;
1610055       //
1610056       // The source address might be zero, to tell
1610057       // that any local
1610058       // address is valid.
1610059       //
1610060       // If it is a TCP/UDP protocol, must have valid
1610061       // ports.
1610062       //
1610063       if (sfd->file->sock->protocol == IPPROTO_TCP
1610064           || sfd->file->sock->protocol == IPPROTO_UDP)
1610065         {
1610066           //
1610067           // Local port.
1610068           //
1610069           if (ntohs (sin->sin_port) == 0)
1610070             {
1610071               //
1610072               // Missing the local port.
1610073               //
1610074               errset (EADDRNOTAVAIL);
1610075               return (-1);
1610076             }
1610077           //
1610078           // If the local port is privileged, must
1610079           // have EUID == 0.
1610080           //
1610081           if (ntohs (sin->sin_port) < 1024)
1610082             {
1610083               if (ps->euid != 0)
1610084                 {
1610085                   //
1610086                   // Missing privileges.
1610087                   //
1610088                   errset (EACCES);
1610089                   return (-1);
1610090                 }
1610091             }
1610092           //
1610093           // If the local port is not given, a default
1610094           // one is assigned.
1610095           //
1610096           if (sfd->file->sock->lport == 0)
1610097             {
1610098               //
1610099               // Must find a free one.
1610100               //
1610101               sfd->file->sock->lport = sock_free_port ();
1610102               if (sfd->file->sock->lport == 0)
1610103                 {
1610104                   //
1610105                   // No port is available.
1610106                   //
1610107                   errset (EAGAIN);
1610108                   return (-1);
1610109                 }
1610110             }
1610111         }
1610112       else
1610113         {
1610114           //
1610115           // Not supported.
1610116           //
1610117           errset (EOPNOTSUPP);
1610118           return (-1);
1610119         }
1610120       //
1610121       // Update the socket.
1610122       //
1610123       sfd->file->sock->family = sin->sin_family;
1610124       sfd->file->sock->laddr = ntohl (sin->sin_addr.s_addr);
1610125       sfd->file->sock->lport = ntohs (sin->sin_port);
1610126 //      sfd->file->sock->bind   = 1;
1610127       //
1610128       // Reset read packets clock time.
1610129       //
1610130       clock_time = s_clock (pid);
1610131       for (i = 0; i < IP_MAX_PACKETS; i++)
1610132         {
1610133           sfd->file->sock->read.clock[i] = clock_time;
1610134         }
1610135     }
1610136   else
1610137     {
1610138       //
1610139       // Unsupported family.
1610140       //
1610141       errset (EAFNOSUPPORT);
1610142       return (-1);
1610143     }
1610144   //
1610145   // Ok.
1610146   //
1610147   return (0);
1610148 }
```


94.8.4 kernel/lib_s/s_brk.c

«

Si veda la sezione 87.5.

```
1620001 #include <errno.h>
1620002 #include <kernel/proc.h>
1620003 #include <kernel/memory.h>
1620004 #include <kernel/lib_k.h>
1620005 #include <kernel/lib_s.h>
1620006 //-----------------------------------------------------------
1620007 #define DEBUG 0
1620008 //-----------------------------------------------------------
1620009 int
1620010 s_brk (pid_t pid, void *address)
1620011 {
1620012   size_t requested_size;
1620013   size_t requested_extra;
1620014   addr_t previous_address_text;
1620015   size_t previous_domain_text;
1620016   addr_t previous_address_data;
1620017   size_t previous_domain_data;
1620018   size_t previous_extra;
1620019   addr_t allocated_text;
```

```
1620020   addr_t allocated_data;
1620021   int status;
1620022   //
1620023   // All segments start form ((void *) 0), so the new
1620024   // address requested is equivalent to the new size
1620025   // for the data segment.
1620026   //
1620027   requested_size = (size_t) address;
1620028   //
1620029   // Check if it is possible to get the new size:
1620030   // cannot be less
1620031   // then the original segment size.
1620032   //
1620033   if (proc_table[pid].domain_data == 0)
1620034     {
1620035       if (proc_table[pid].domain_text > requested_size)
1620036         {
1620037           requested_extra = 0;
1620038         }
1620039       else
1620040         {
1620041           requested_extra = requested_size
1620042             - proc_table[pid].domain_text;
1620043         }
1620044     }
1620045   else
1620046     {
1620047       if (proc_table[pid].domain_data > requested_size)
1620048         {
1620049           requested_extra = 0;
1620050         }
1620051       else
1620052         {
1620053           requested_extra = requested_size
1620054             - proc_table[pid].domain_data;
1620055         }
1620056     }
1620057   //
1620058   // Now make shure that the new value is a multiple
1620059   // of
1620060   // MEM_BLOCK_SIZE!
1620061   //
1620062   if (requested_extra % MEM_BLOCK_SIZE)
1620063     {
1620064       requested_extra =
1620065         (((requested_extra / MEM_BLOCK_SIZE) +
1620066           1) * MEM_BLOCK_SIZE);
1620067     }
1620068   //
1620069   // Now resize the process.
1620070   //
1620071   if (requested_extra == proc_table[pid].extra_data)
1620072     {
1620073       //
1620074       // Nothing have to change.
1620075       //
1620076       return (0);
1620077     }
1620078   else if (requested_extra < proc_table[pid].extra_data)
1620079     {
1620080       //
1620081       // Just reduce.
1620082       //
1620083       previous_extra = proc_table[pid].extra_data;
1620084       proc_table[pid].extra_data = requested_extra;
1620085       //
1620086       // Update process DATA segment inside the GDT
1620087       // table.
1620088       //
1620089       if (proc_table[pid].domain_data > 0)
1620090         {
1620091           gdt_segment (gdt_pid_to_segment_data (pid),
1620092                        (uint32_t)
1620093                        proc_table[pid].address_data,
1620094                        (uint32_t) ((proc_table
1620095                                     [pid].domain_data +
1620096                                     proc_table
1620097                                     [pid].extra_data) /
1620098                                    MEM_BLOCK_SIZE), 1, 0,
1620099                        0);
1620100         }
1620101       else
1620102         {
1620103           gdt_segment (gdt_pid_to_segment_data (pid),
1620104                        (uint32_t)
1620105                        proc_table[pid].address_text,
1620106                        (uint32_t) ((proc_table
1620107                                     [pid].domain_text +
1620108                                     proc_table
1620109                                     [pid].extra_data) /
1620110                                    MEM_BLOCK_SIZE), 1, 0,
1620111                        0);
1620112         }
1620113       //
1620114       // Release memory.
1620115       //
1620116       if (proc_table[pid].domain_data > 0)
1620117         {
1620118           status =
1620119             mb_reduce ((proc_table[pid].address_data +
1620120                         proc_table[pid].domain_data),
1620121                        proc_table[pid].extra_data,
1620122                        previous_extra);
1620123         }
1620124       else
1620125         {
1620126           status =
1620127             mb_reduce ((proc_table[pid].address_text +
1620128                         proc_table[pid].domain_text),
1620129                        proc_table[pid].extra_data,
1620130                        previous_extra);
1620131         }
1620132       //
1620133       if (status < 0)
1620134         {
1620135           //
1620136           // What happened?
1620137           //
1620138           k_perror (NULL);
1620139         }
1620140     }
1620141   else
1620142     {
1620143       //
1620144       // A bigger size was requested. Save previous
1620145       // information.
1620146       //
1620147       previous_address_text = proc_table[pid].address_text;
1620148       previous_domain_text = proc_table[pid].domain_text;
1620149       previous_address_data = proc_table[pid].address_data;
1620150       previous_domain_data = proc_table[pid].domain_data;
1620151       previous_extra = proc_table[pid].extra_data;
1620152       //
1620153       // Allocate memory for text,
1620154       // if text and data are inside the same address
1620155       // space;
1620156       // otherwise only the data segment is involved.
1620157       //
1620158       if (proc_table[pid].domain_data == 0)
1620159         {
1620160           allocated_text =
1620161             mb_alloc_size (proc_table[pid].domain_text +
1620162                            requested_extra);
1620163           //
1620164           if (allocated_text == 0)
1620165             {
1620166               errset (ENOMEM);  // Not enough space.
1620167               return (-1);
1620168             }
1620169           //
1620170           if (DEBUG)
1620171             {
1620172               k_printf ("%s:%i:mb_alloc_size(%zi)",
1620173                         __FILE__, __LINE__,
1620174                         (proc_table[pid].domain_text
1620175                          + requested_extra));
1620176             }
1620177         }
1620178       //
1620179       // Allocate memory for data, if necessary.
1620180       //
1620181       if (proc_table[pid].domain_data > 0)
1620182         {
1620183           allocated_data =
1620184             mb_alloc_size (proc_table[pid].domain_data +
1620185                            requested_extra);
1620186           //
1620187           if (allocated_data == 0)
1620188             {
1620189               //
1620190               // Please note that, if we are here, no
1620191               // memory
1620192               // for the text was allocated!
1620193               //
```

```
1620194               errset (ENOMEM);  // Not enough space.
1620195               return (-1);
1620196             }
1620197           //
1620198           if (DEBUG)
1620199             {
1620200               k_printf ("%s:%i:mb_alloc_size(%zi)",
1620201                         __FILE__, __LINE__,
1620202                         (proc_table[pid].domain_data
1620203                          + requested_extra));
1620204             }
1620205         }
1620206       //
1620207       // Copy the process text and, data in memory: if
1620208       // size is zero, no copy is made. But the text
1620209       // is
1620210       // copied only if text and data live together.
1620211       //
1620212       if (proc_table[pid].domain_data == 0)
1620213         {
1620214           memcpy ((void *) allocated_text,
1620215                   (void *) proc_table[pid].address_text,
1620216                   (size_t) (proc_table[pid].domain_text +
1620217                             proc_table[pid].extra_data));
1620218         }
1620219       else
1620220         {
1620221           memcpy ((void *) allocated_data,
1620222                   (void *) proc_table[pid].address_data,
1620223                   (size_t) (proc_table[pid].domain_data +
1620224                             proc_table[pid].extra_data));
1620225         }
1620226       //
1620227       // Update process information.
1620228       //
1620229       if (proc_table[pid].domain_data == 0)
1620230         {
1620231           proc_table[pid].address_text = allocated_text;
1620232         }
1620233       else
1620234         {
1620235           proc_table[pid].address_data = allocated_data;
1620236         }
1620237       proc_table[pid].extra_data = requested_extra;
1620238       //
1620239       // Update process TEXT segment inside the GDT
1620240       // table.
1620241       //
1620242       gdt_segment (gdt_pid_to_segment_text (pid),
1620243                    (uint32_t) proc_table[pid].address_text,
1620244                    (uint32_t) (proc_table[pid].domain_text /
1620245                                MEM_BLOCK_SIZE), 1, 1, 0);
1620246       //
1620247       // Update process DATA segment inside the GDT
1620248       // table.
1620249       //
1620250       if (proc_table[pid].domain_data > 0)
1620251         {
1620252           gdt_segment (gdt_pid_to_segment_data (pid),
1620253                        (uint32_t)
1620254                        proc_table[pid].address_data,
1620255                        (uint32_t) ((proc_table
1620256                                     [pid].domain_data +
1620257                                     proc_table
1620258                                     [pid].extra_data) /
1620259                                    MEM_BLOCK_SIZE), 1, 0,
1620260                        0);
1620261         }
1620262       else
1620263         {
1620264           gdt_segment (gdt_pid_to_segment_data (pid),
1620265                        (uint32_t)
1620266                        proc_table[pid].address_text,
1620267                        (uint32_t) ((proc_table
1620268                                     [pid].domain_text +
1620269                                     proc_table
1620270                                     [pid].extra_data) /
1620271                                    MEM_BLOCK_SIZE), 1, 0,
1620272                        0);
1620273         }
1620274       //
1620275       // Now release the old memory!
1620276       //
1620277       if (proc_table[pid].domain_data == 0)
1620278         {
1620279           mb_free (previous_address_text,
1620280                    previous_domain_text + previous_extra);
```



```
1620281         }
1620282       else
1620283         {
1620284           mb_free (previous_address_data,
1620285                    previous_domain_data + previous_extra);
1620286         }
1620287     }
1620288   //
1620289   // Ok.
1620290   //
1620291   return (0);
1620292 }
```

94.8.5 kernel/lib_s/s_chdir.c

Si veda la sezione 87.6.

```
1630001 #include <kernel/fs.h>
1630002 #include <errno.h>
1630003 #include <kernel/proc.h>
1630004 #include <kernel/lib_s.h>
1630005 //-----------------------------------------------------------
1630006 int
1630007 s_chdir (pid_t pid, const char *path)
1630008 {
1630009   proc_t *ps;
1630010   inode_t *inode_directory;
1630011   int status;
1630012   char path_directory[PATH_MAX];
1630013   //
1630014   // Get process.
1630015   //
1630016   ps = proc_reference (pid);
1630017   //
1630018   // The full directory path is needed.
1630019   //
1630020   status = path_full (path, ps->path_cwd, path_directory);
1630021   if (status < 0)
1630022     {
1630023       return (-1);
1630024     }
1630025   //
1630026   // Try to load the new directory inode.
1630027   //
1630028   inode_directory = path_inode (pid, path_directory);
1630029   if (inode_directory == NULL)
1630030     {
1630031       //
1630032       // Cannot access the directory: it does not
1630033       // exists or
1630034       // permissions are not sufficient. Variable
1630035       // 'errno' is set by
1630036       // function 'inode_directory()'.
1630037       //
1630038       errset (errno);
1630039       return (-1);
1630040     }
1630041   //
1630042   // Inode loaded: release the old directory and set
1630043   // the new one.
1630044   //
1630045   inode_put (ps->inode_cwd);
1630046   //
1630047   ps->inode_cwd = inode_directory;
1630048   strncpy (ps->path_cwd, path_directory, PATH_MAX);
1630049   //
1630050   // Return.
1630051   //
1630052   return (0);
1630053 }
```

94.8.6 kernel/lib_s/s_chmod.c

Si veda la sezione 87.7.

```
1640001 #include <kernel/fs.h>
1640002 #include <errno.h>
1640003 #include <kernel/proc.h>
1640004 #include <kernel/lib_s.h>
1640005 //-----------------------------------------------------------
1640006 int
1640007 s_chmod (pid_t pid, const char *path, mode_t mode)
1640008 {
1640009   proc_t *ps;
1640010   inode_t *inode;
1640011   //
1640012   // Get process.
```

```
1640013   //
1640014   ps = proc_reference (pid);
1640015   //
1640016   // Try to load the file inode.
1640017   //
1640018   inode = path_inode (pid, path);
1640019   if (inode == NULL)
1640020     {
1640021       //
1640022       // Cannot access the file: it does not exists or
1640023       // permissions are
1640024       // not sufficient. Variable 'errno' is set by
1640025       // function
1640026       // 'inode_directory()'.
1640027       //
1640028       return (-1);
1640029     }
1640030   //
1640031   // Verify to be root or to be the owner.
1640032   //
1640033   if (ps->euid != 0 && ps->euid != inode->uid)
1640034     {
1640035       errset (EACCES);  // Permission denied.
1640036       return (-1);
1640037     }
1640038   //
1640039   // Update the mode: the file type is kept and the
1640040   // rest is taken form the parameter 'mode'.
1640041   //
1640042   inode->mode = (S_IFMT & inode->mode) | (~S_IFMT & mode);
1640043   //
1640044   // Save and release the inode.
1640045   //
1640046   inode->changed = 1;
1640047   inode_save (inode);
1640048   inode_put (inode);
1640049   //
1640050   // Return.
1640051   //
1640052   return (0);
1640053 }
```

94.8.7 kernel/lib_s/s_chown.c

Si veda la sezione 87.8.

```
1650001 #include <kernel/fs.h>
1650002 #include <errno.h>
1650003 #include <kernel/proc.h>
1650004 #include <kernel/lib_s.h>
1650005 //-------------------------------------------------------
1650006 int
1650007 s_chown (pid_t pid, const char *path, uid_t uid, gid_t gid)
1650008 {
1650009   proc_t *ps;
1650010   inode_t *inode;
1650011   //
1650012   // Get process.
1650013   //
1650014   ps = proc_reference (pid);
1650015   //
1650016   // Must be root, as the ability to change group is
1650017   // not considered.
1650018   //
1650019   if (ps->euid != 0)
1650020     {
1650021       errset (EPERM);   // Operation not permitted.
1650022       return (-1);
1650023     }
1650024   //
1650025   // Try to load the file inode.
1650026   //
1650027   inode = path_inode (pid, path);
1650028   if (inode == NULL)
1650029     {
1650030       //
1650031       // Cannot access the file: it does not exists or
1650032       // permissions are
1650033       // not sufficient. Variable 'errno' is set by
1650034       // function
1650035       // 'inode_directory()'.
1650036       //
1650037       return (-1);
1650038     }
1650039   //
1650040   // Update the owner and group.
1650041   //
```



```
1650042   if (uid != -1)
1650043     {
1650044       inode->uid = uid;
1650045       inode->changed = 1;
1650046     }
1650047   if (gid != -1)
1650048     {
1650049       inode->gid = gid;
1650050       inode->changed = 1;
1650051     }
1650052   //
1650053   // Save and release the inode.
1650054   //
1650055   inode_save (inode);
1650056   inode_put (inode);
1650057   //
1650058   // Return.
1650059   //
1650060   return (0);
1650061 }
```

94.8.8 kernel/lib_s/s_clock.c

Si veda la sezione 87.9.

```
1660001 #include <kernel/lib_s.h>
1660002 //-------------------------------------------------------
1660003 extern clock_t _clock_kernel;   // uint64_t
1660004 //-------------------------------------------------------
1660005 clock_t
1660006 s_clock (pid_t pid)
1660007 {
1660008   return (_clock_kernel);
1660009 }
```

94.8.9 kernel/lib_s/s_close.c

Si veda la sezione 87.10.

```
1670001 #include <kernel/proc.h>
1670002 #include <fcntl.h>
1670003 #include <errno.h>
1670004 //-------------------------------------------------------
1670005 int
1670006 s_close (pid_t pid, int fdn)
1670007 {
1670008   fd_t *fd;
1670009   int status;
1670010   //
1670011   // Get file descriptor.
1670012   //
1670013   fd = fd_reference (pid, &fdn);
1670014   if (fd == NULL || fd->file == NULL
1670015       || (fd->file->inode == NULL
1670016           && fd->file->sock == NULL))
1670017     {
1670018       errset (EBADF);   // Bad file descriptor.
1670019       return (-1);
1670020     }
1670021   //
1670022   //
1670023   //
1670024   if (fd->file->inode != NULL)  // Inode
1670025     {
1670026       //
1670027       // File descriptor with inode.
1670028       //
1670029       // If it is a pipe, some special things must be
1670030       // done.
1670031       //
1670032       if (S_ISFIFO (fd->file->inode->mode))
1670033         {
1670034           if (fd->fl_flags & O_RDONLY)
1670035             {
1670036               fd->file->inode->pipe_ref_read--;
1670037               if (fd->file->inode->pipe_ref_read == 0)
1670038                 {
1670039                   proc_wakeup_pipe_write (fd->file->inode);
1670040                 }
1670041             }
1670042           //
1670043           if (fd->fl_flags & O_WRONLY)
1670044             {
1670045               fd->file->inode->pipe_ref_write--;
1670046               if (fd->file->inode->pipe_ref_write == 0)
1670047                 {
1670048                   proc_wakeup_pipe_read (fd->file->inode);
```

```
1670049                 }
1670050               }
1670051           }
1670052       }
1670053     else  // Socket
1670054       {
1670055         //
1670056         // File descriptor with socket.
1670057         //
1670058         status = tcp_close (fd->file->sock);
1670059         if (status < 0
1670060             && (errno == EINPROGRESS || errno == EALREADY))
1670061           {
1670062             errset (errno);
1670063             return (status);
1670064           }
1670065         //
1670066         // Otherwise, the socket is closed and can
1670067         // continue
1670068         // with the other references.
1670069         //
1670070       }
1670071     //
1670072     // Reduce references inside the file table item
1670073     // and remove item if it reaches zero.
1670074     //
1670075     fd->file->references--;
1670076     if (fd->file->references == 0)
1670077       {
1670078         fd->file->oflags = 0;
1670079         fd->file->inode = NULL;
1670080         //
1670081         // Put inode, or release the socket, because
1670082         // there are no more
1670083         // file references.
1670084         //
1670085         if (fd->file->inode != NULL)
1670086           {
1670087             inode_put (fd->file->inode);
1670088           }
1670089         if (fd->file->sock != NULL)
1670090           {
1670091             sock_put (fd->file->sock);
1670092           }
1670093       }
1670094     //
1670095     // Remove file descriptor.
1670096     //
1670097     fd->fl_flags = 0;
1670098     fd->fd_flags = 0;
1670099     fd->file = NULL;
1670100     //
1670101     //
1670102     //
1670103     return (0);
1670104 }
```

94.8.10 kernel/lib_s/s_connect.c

«

Si veda la sezione 87.11.

```
1680001 #include <kernel/proc.h>
1680002 #include <kernel/lib_s.h>
1680003 #include <kernel/lib_k.h>
1680004 #include <kernel/net/route.h>
1680005 #include <errno.h>
1680006 #include <fcntl.h>
1680007 #include <sys/socket.h>
1680008 #include <arpa/inet.h>
1680009 //-----------------------------------------------------------
1680010 int
1680011 s_connect (pid_t pid, int sfdn,
1680012           const struct sockaddr *addr, socklen_t addrlen)
1680013 {
1680014   fd_t *sfd;
1680015   struct sockaddr_in *sin;
1680016   clock_t clock_time;
1680017   int i;
1680018   int status;
1680019   //
1680020   // Get file descriptor and verify that it is a
1680021   // socket.
1680022   //
1680023   sfd = fd_reference (pid, &sfdn);
1680024   if (sfd == NULL || sfd->file == NULL)
1680025     {
1680026       errset (EBADF);   // Bad file descriptor.
```



```
1680027       return (-1);
1680028     }
1680029   if (sfd->file->sock == NULL)
1680030     {
1680031       errset (ENOTSOCK);        // Not a socket.
1680032       return (-1);
1680033     }
1680034   //
1680035   // Verify to have a valid address pointer.
1680036   //
1680037   if (addr == NULL)
1680038     {
1680039       errset (EINVAL);
1680040       return (-1);
1680041     }
1680042   //
1680043   // Check minimal address size.
1680044   //
1680045   if (addrlen < sizeof (struct sockaddr))
1680046     {
1680047       errset (EINVAL);
1680048       return (-1);
1680049     }
1680050   //
1680051   //
1680052   //
1680053   if (addr->sa_family == AF_INET)
1680054     {
1680055       sin = (struct sockaddr_in *) addr;
1680056       //
1680057       // Check to have the destination IP address.
1680058       //
1680059       if (sin->sin_addr.s_addr == 0)
1680060         {
1680061           //
1680062           // This is not valid.
1680063           //
1680064           errset (EADDRNOTAVAIL);
1680065           return (-1);
1680066         }
1680067       //
1680068       // If it is a TCP/UDP protocol, must have valid
1680069       // ports.
1680070       //
1680071       if (sfd->file->sock->protocol == IPPROTO_TCP
1680072           || sfd->file->sock->protocol == IPPROTO_UDP)
1680073         {
1680074           //
1680075           // Remote port.
1680076           //
1680077           if (sin->sin_port == 0)
1680078             {
1680079               //
1680080               // Missing the remote port.
1680081               //
1680082               errset (EADDRNOTAVAIL);
1680083               return (-1);
1680084             }
1680085           //
1680086           // Local port.
1680087           //
1680088           if (sfd->file->sock->lport == 0)
1680089             {
1680090               //
1680091               // Must find a free one.
1680092               //
1680093               sfd->file->sock->lport = sock_free_port ();
1680094               if (sfd->file->sock->lport == 0)
1680095                 {
1680096                   //
1680097                   // No port is available.
1680098                   //
1680099                   errset (EAGAIN);
1680100                   return (-1);
1680101                 }
1680102             }
1680103         }
1680104       //
1680105       // Update the socket, but not a TCP connection
1680106       // already
1680107       // working (TCP not connected has a zeroed
1680108       // 'tcp.conn' filled).
1680109       //
1680110       if (sfd->file->sock->tcp.conn == 0
1680111           || sfd->file->sock->tcp.conn == TCP_CLOSE)
1680112         {
1680113           //
```

```
1680114           // Update the socket.
1680115           //
1680116           sfd->file->sock->family = sin->sin_family;
1680117           sfd->file->sock->raddr =
1680118             ntohl (sin->sin_addr.s_addr);
1680119           sfd->file->sock->rport = ntohs (sin->sin_port);
1680120           //
1680121           // Reset read packets clock time.
1680122           //
1680123           clock_time = s_clock (pid);
1680124           for (i = 0; i < IP_MAX_PACKETS; i++)
1680125             {
1680126               sfd->file->sock->read.clock[i] = clock_time;
1680127             }
1680128         }
1680129       else
1680130         {
1680131           //
1680132           // It *is* a TCP connection already working:
1680133           // verify that
1680134           // the socket is set as expected
1680135           //
1680136           if (sfd->file->sock->family !=
1680137               sin->sin_family
1680138               || sfd->file->sock->raddr !=
1680139               ntohl (sin->sin_addr.s_addr)
1680140               || sfd->file->sock->rport !=
1680141               ntohs (sin->sin_port))
1680142             {
1680143               //
1680144               // The socket address is changed!
1680145               //
1680146               errset (EISCONN);
1680147               return (-1);
1680148             }
1680149         }
1680150       //
1680151       // If it is a TCP, not already working, should
1680152       // connect.
1680153       // The function 'tcp_connect()' will verify the
1680154       // connection
1680155       // status.
1680156       //
1680157       if (sfd->file->sock->protocol == IPPROTO_TCP)
1680158         {
1680159           //
1680160           // Be shure to have a source address.
1680161           //
1680162           if (sfd->file->sock->laddr == 0)
1680163             {
1680164               //
1680165               // Default source address: get the
1680166               // source address from the
1680167               // routing table, based on the
1680168               // destination.
1680169               //
1680170               sfd->file->sock->laddr
1680171                 =
1680172                 route_remote_to_local (sfd->file->
1680173                                        sock->raddr);
1680174               if (sfd->file->sock->laddr ==
1680175                   ((h_addr_t) - 1))
1680176                 {
1680177                   errset (errno);
1680178                   return (-1);
1680179                 }
1680180             }
1680181           //
1680182           // Call tcp_connect ().
1680183           //
1680184           status = tcp_connect (sfd->file->sock);
1680185           if (status)
1680186             {
1680187               errset (errno);
1680188             }
1680189           return (status);
1680190         }
1680191     }
1680192   else
1680193     {
1680194       //
1680195       // It is not AF_INET: unsupported address
1680196       // family.
1680197       //
1680198       errset (EAFNOSUPPORT);
1680199       return (-1);
1680200     }
```



```
1680201   //
1680202   // Ok.
1680203   //
1680204   return (0);
1680205 }
```

94.8.11 kernel/lib_s/s_dup.c

Si veda la sezione 87.12.

```
1690001 #include <kernel/lib_s.h>
1690002 #include <kernel/fs.h>
1690003 //-------------------------------------------------------------
1690004 int
1690005 s_dup (pid_t pid, int fdn_old)
1690006 {
1690007   return (fd_dup (pid, fdn_old, 0));
1690008 }
```

94.8.12 kernel/lib_s/s_dup2.c

Si veda la sezione 87.12.

```
1700001 #include <kernel/proc.h>
1700002 #include <kernel/lib_s.h>
1700003 #include <errno.h>
1700004 #include <fcntl.h>
1700005 //-------------------------------------------------------------
1700006 int
1700007 s_dup2 (pid_t pid, int fdn_old, int fdn_new)
1700008 {
1700009   proc_t *ps;
1700010   int status;
1700011   //
1700012   // Get process.
1700013   //
1700014   ps = proc_reference (pid);
1700015   //
1700016   // Verify if 'fdn_old' is a valid value.
1700017   //
1700018   if (fdn_old < 0 ||
1700019       fdn_old >= OPEN_MAX || ps->fd[fdn_old].file == NULL)
1700020     {
1700021       errset (EBADF);   // Bad file descriptor.
1700022       return (-1);
1700023     }
1700024   //
1700025   // Check if 'fd_old' and 'fd_new' are the same.
1700026   //
1700027   if (fdn_old == fdn_new)
1700028     {
1700029       return (fdn_new);
1700030     }
1700031   //
1700032   // Close 'fdn_new' if it is open and copy 'fdn_old'
1700033   // into it.
1700034   //
1700035   if (ps->fd[fdn_new].file != NULL)
1700036     {
1700037       status = s_close (pid, fdn_new);
1700038       if (status != 0)
1700039         {
1700040           return (-1);
1700041         }
1700042     }
1700043   ps->fd[fdn_new].fl_flags = ps->fd[fdn_old].fl_flags;
1700044   ps->fd[fdn_new].fd_flags =
1700045     ps->fd[fdn_old].fd_flags & ~FD_CLOEXEC;
1700046   ps->fd[fdn_new].file = ps->fd[fdn_old].file;
1700047   ps->fd[fdn_new].file->references++;
1700048   return (fdn_new);
1700049 }
```

94.8.13 kernel/lib_s/s_fchmod.c

Si veda la sezione 87.7.

```
1710001 #include <kernel/proc.h>
1710002 #include <kernel/lib_s.h>
1710003 #include <sys/stat.h>
1710004 #include <errno.h>
1710005 //-------------------------------------------------------------
1710006 int
1710007 s_fchmod (pid_t pid, int fdn, mode_t mode)
1710008 {
1710009   proc_t *ps;
1710010   inode_t *inode;
```

```
1710011   //
1710012   // Get process.
1710013   //
1710014   ps = proc_reference (pid);
1710015   //
1710016   // Verify if the file descriptor is valid.
1710017   //
1710018   if (ps->fd[fdn].file == NULL)
1710019     {
1710020       errset (EBADF);   // Bad file descriptor.
1710021       return (-1);
1710022     }
1710023   //
1710024   // Reach the inode.
1710025   //
1710026   inode = ps->fd[fdn].file->inode;
1710027   //
1710028   // If the Inode does not exist, exit with error.
1710029   //
1710030   if (inode == NULL)
1710031     {
1710032       errset (ENOENT);
1710033       return (-1);
1710034     }
1710035   //
1710036   // Verify to be the owner, or at least to be UID ==
1710037   // 0.
1710038   //
1710039   if (ps->euid != inode->uid && ps->euid != 0)
1710040     {
1710041       errset (EACCES);  // Permission denied.
1710042       return (-1);
1710043     }
1710044   //
1710045   // Update the mode: the file type is kept and the
1710046   // rest is taken form the parameter 'mode'.
1710047   //
1710048   inode->mode = (S_IFMT & inode->mode) | (~S_IFMT & mode);
1710049   //
1710050   // Save the inode.
1710051   //
1710052   inode->changed = 1;
1710053   inode_save (inode);
1710054   //
1710055   // Return.
1710056   //
1710057   return (0);
1710058 }
```

94.8.14 kernel/lib_s/s_fchown.c

«

Si veda la sezione 87.8.

```
1720001 #include <kernel/proc.h>
1720002 #include <kernel/lib_s.h>
1720003 #include <errno.h>
1720004 //-----------------------------------------------------------
1720005 int
1720006 s_fchown (pid_t pid, int fdn, uid_t uid, gid_t gid)
1720007 {
1720008   proc_t *ps;
1720009   inode_t *inode;
1720010   //
1720011   // Get process.
1720012   //
1720013   ps = proc_reference (pid);
1720014   //
1720015   // Verify if the file descriptor is valid.
1720016   //
1720017   if (ps->fd[fdn].file == NULL)
1720018     {
1720019       errset (EBADF);   // Bad file descriptor.
1720020       return (-1);
1720021     }
1720022   //
1720023   // Reach the inode.
1720024   //
1720025   inode = ps->fd[fdn].file->inode;
1720026   //
1720027   // If the Inode does not exist, exit with error.
1720028   //
1720029   if (inode == NULL)
1720030     {
1720031       errset (ENOENT);
1720032       return (-1);
1720033     }
1720034   //
```



```
1720035   // Verify to be root, as the ability to change group
1720036   // is not taken into consideration.
1720037   //
1720038   if (ps->euid != 0)
1720039     {
1720040       errset (EACCES);  // Permission denied.
1720041       return (-1);
1720042     }
1720043   //
1720044   // Update the ownership.
1720045   //
1720046   if (uid != -1)
1720047     {
1720048       inode->uid = uid;
1720049       inode->changed = 1;
1720050     }
1720051   if (gid != -1)
1720052     {
1720053       inode->gid = gid;
1720054       inode->changed = 1;
1720055     }
1720056   //
1720057   // Save the inode.
1720058   //
1720059   inode->changed = 1;
1720060   inode_save (inode);
1720061   //
1720062   // Return.
1720063   //
1720064   return (0);
1720065 }
```

94.8.15 kernel/lib_s/s_fcntl.c

«

Si veda la sezione 87.18.

```
1730001 #include <kernel/proc.h>
1730002 #include <kernel/lib_s.h>
1730003 #include <kernel/fs.h>
1730004 #include <errno.h>
1730005 #include <fcntl.h>
1730006 //-----------------------------------------------------------
1730007 int
1730008 s_fcntl (pid_t pid, int fdn, int cmd, int arg)
1730009 {
1730010   proc_t *ps;
1730011   int mask;
1730012   //
1730013   // Get process.
1730014   //
1730015   ps = proc_reference (pid);
1730016   //
1730017   // Verify if the file descriptor is valid.
1730018   //
1730019   if (ps->fd[fdn].file == NULL)
1730020     {
1730021       errset (EBADF);   // Bad file descriptor.
1730022       return (-1);
1730023     }
1730024   //
1730025   //
1730026   //
1730027   switch (cmd)
1730028     {
1730029     case F_DUPFD:
1730030       return (fd_dup (pid, fdn, arg));
1730031       break;
1730032     case F_GETFD:
1730033       return (ps->fd[fdn].fd_flags);
1730034       break;
1730035     case F_SETFD:
1730036       ps->fd[fdn].fd_flags = arg;
1730037       return (0);
1730038     case F_GETFL:
1730039       return (ps->fd[fdn].fl_flags);
1730040     case F_SETFL:
1730041       //
1730042       // Calculate a mask with bits that are *not* to
1730043       // be set.
1730044       //
1730045       mask =
1730046         (O_ACCMODE | O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);
1730047       //
1730048       // Set to zero the bits that are not to be set
1730049       // from
1730050       // the argument.
1730051       //
```

```
1730052         arg = (arg & ~mask);
1730053         //
1730054         // Set to zero the bit that *are* to be set.
1730055         //
1730056         ps->fd[fdn].fl_flags &= mask;
1730057         //
1730058         // Set the bits, already filtered inside the
1730059         // argument.
1730060         //
1730061         ps->fd[fdn].fl_flags |= arg;
1730062         //
1730063         return (0);
1730064       default:
1730065         errset (EINVAL);  // Not implemented.
1730066         return (-1);
1730067       }
1730068   }
```

94.8.16 kernel/lib_s/s_fork.c

«

Si veda la sezione 87.19.

```
1740001 #include <kernel/proc.h>
1740002 #include <errno.h>
1740003 #include <fcntl.h>
1740004 #include <kernel/lib_k.h>
1740005 #include <kernel/lib_s.h>
1740006 //-----------------------------------------------------------
1740007 #define DEBUG 0
1740008 //-----------------------------------------------------------
1740009 extern uint32_t proc_stack_pointer;
1740010 //-----------------------------------------------------------
1740011 pid_t
1740012 s_fork (pid_t ppid)
1740013 {
1740014   pid_t pid;
1740015   pid_t zombie;
1740016   addr_t allocated_text = 0;
1740017   addr_t allocated_data = 0;
1740018   addr_t addr_stack_pointer = 0;
1740019   int fdn;
1740020   uint16_t segment_descriptor;
1740021   int sig;
1740022   //
1740023   // Find a free PID.
1740024   //
1740025   for (pid = 1; pid < PROCESS_MAX; pid++)
1740026     {
1740027       if (proc_table[pid].status == PROC_EMPTY)
1740028         {
1740029           break;
1740030         }
1740031     }
1740032   if (pid >= PROCESS_MAX)
1740033     {
1740034       //
1740035       // There is no free pid.
1740036       //
1740037       errset (ENOMEM);  // Not enough space.
1740038       return (-1);
1740039     }
1740040   //
1740041   // Before allocating a new process, must check if
1740042   // there are some
1740043   // zombie slots, still with original segment data:
1740044   // should reset
1740045   // them now!
1740046   //
1740047   for (zombie = 1; zombie < PROCESS_MAX; zombie++)
1740048     {
1740049       if (proc_table[zombie].status == PROC_ZOMBIE
1740050           && (proc_table[zombie].address_text != 0
1740051               || proc_table[zombie].domain_text != 0))
1740052         {
1740053           proc_table[zombie].address_text = (addr_t) 0;
1740054           proc_table[zombie].domain_text = (size_t) 0;
1740055           proc_table[zombie].address_data = (addr_t) 0;
1740056           proc_table[zombie].domain_data = (size_t) 0;
1740057           proc_table[zombie].domain_stack = (size_t) 0;
1740058           proc_table[zombie].extra_data = (size_t) 0;
1740059           proc_table[zombie].sp = 0;
1740060         }
1740061     }
1740062   //
1740063   // Allocate memory for text, if text and data are
1740064   // inside
1740065   // the same address space.
```



```
1740066   //
1740067   if (proc_table[ppid].domain_data == 0)
1740068     {
1740069       allocated_text =
1740070         mb_alloc_size (proc_table[ppid].domain_text +
1740071                        proc_table[ppid].extra_data);
1740072       //
1740073       if (allocated_text == 0)
1740074         {
1740075           errset (ENOMEM);      // Not enough space.
1740076           return ((pid_t) - 1);
1740077         }
1740078       //
1740079       if (DEBUG)
1740080         {
1740081           k_printf ("%s:%i:mb_alloc_size(%zi)",
1740082                     __FILE__, __LINE__,
1740083                     (proc_table[ppid].domain_text
1740084                      + proc_table[ppid].extra_data));
1740085         }
1740086     }
1740087   //
1740088   // Allocate memory for data, if necessary.
1740089   //
1740090   if (proc_table[ppid].domain_data > 0)
1740091     {
1740092       allocated_data =
1740093         mb_alloc_size (proc_table[ppid].domain_data +
1740094                        proc_table[ppid].extra_data);
1740095       //
1740096       if (allocated_data == 0)
1740097         {
1740098           //
1740099           // Please note that, if we are here, no
1740100           // memory
1740101           // for the text was allocated!
1740102           //
1740103           errset (ENOMEM);      // Not enough space.
1740104           return ((pid_t) - 1);
1740105         }
1740106       //
1740107       if (DEBUG)
1740108         {
1740109           k_printf ("%s:%i:mb_alloc_size(%zi)",
1740110                     __FILE__, __LINE__,
1740111                     (proc_table[ppid].domain_data
1740112                      + proc_table[ppid].extra_data));
1740113         }
1740114     }
1740115   //
1740116   // Copy the process text and, data in memory: if
1740117   // size is zero, no copy is made. But the text is
1740118   // copied only if text and data live together.
1740119   //
1740120   if (proc_table[ppid].domain_data == 0)
1740121     {
1740122       memcpy ((void *) allocated_text,
1740123               (void *) proc_table[ppid].address_text,
1740124               (size_t) (proc_table[ppid].domain_text
1740125                         + proc_table[ppid].extra_data));
1740126     }
1740127   else
1740128     {
1740129       memcpy ((void *) allocated_data,
1740130               (void *) proc_table[ppid].address_data,
1740131               (size_t) (proc_table[ppid].domain_data
1740132                         + proc_table[ppid].extra_data));
1740133     }
1740134   //
1740135   // Allocate the new PID inside the 'proc_table[]'.
1740136   //
1740137   proc_table[pid].ppid = ppid;
1740138   proc_table[pid].pgrp = proc_table[ppid].pgrp;
1740139   proc_table[pid].uid = proc_table[ppid].uid;
1740140   proc_table[pid].euid = proc_table[ppid].euid;
1740141   proc_table[pid].suid = proc_table[ppid].suid;
1740142   proc_table[pid].gid = proc_table[ppid].gid;
1740143   proc_table[pid].egid = proc_table[ppid].egid;
1740144   proc_table[pid].sgid = proc_table[ppid].sgid;
1740145   proc_table[pid].device_tty = proc_table[ppid].device_tty;
1740146   proc_table[pid].sig_status = 0;
1740147   proc_table[pid].sig_ignore = 0;
1740148   //
1740149   for (sig = 0; sig < MAX_SIGNALS; sig++)
1740150     {
1740151       proc_table[pid].sig_handler[sig]
1740152         = proc_table[ppid].sig_handler[sig];
```

```
1740153     }
1740154   //
1740155   proc_table[pid].usage = 0;
1740156   proc_table[pid].status = PROC_CREATED;
1740157   proc_table[pid].wakeup_events = 0;
1740158   proc_table[pid].wakeup_signal = 0;
1740159   proc_table[pid].wakeup_timer = 0;
1740160   //
1740161   if (proc_table[ppid].domain_data != 0)
1740162     {
1740163       proc_table[pid].address_text =
1740164         proc_table[ppid].address_text;
1740165     }
1740166   else
1740167     {
1740168       proc_table[pid].address_text = allocated_text;
1740169     }
1740170   proc_table[pid].domain_text =
1740171     proc_table[ppid].domain_text;
1740172   proc_table[pid].address_data = allocated_data;
1740173   proc_table[pid].domain_data =
1740174     proc_table[ppid].domain_data;
1740175   proc_table[pid].domain_stack =
1740176     proc_table[ppid].domain_stack;
1740177   proc_table[pid].extra_data = proc_table[ppid].extra_data;
1740178   proc_table[pid].sp = proc_stack_pointer;
1740179   proc_table[pid].ret = 0;
1740180   proc_table[pid].inode_cwd = proc_table[ppid].inode_cwd;
1740181   proc_table[pid].umask = proc_table[ppid].umask;
1740182   strncpy (proc_table[pid].name, proc_table[ppid].name,
1740183            PATH_MAX);
1740184   strncpy (proc_table[pid].path_cwd,
1740185            proc_table[ppid].path_cwd, PATH_MAX);
1740186   //
1740187   // Update process TEXT segment inside the GDT table.
1740188   //
1740189   gdt_segment (gdt_pid_to_segment_text (pid),
1740190                (uint32_t) proc_table[pid].address_text,
1740191                (uint32_t) (proc_table[pid].domain_text /
1740192                            4096), 1, 1, 0);
1740193   //
1740194   // Update process DATA segment inside the GDT table.
1740195   //
1740196   if (proc_table[pid].domain_data > 0)
1740197     {
1740198       gdt_segment (gdt_pid_to_segment_data (pid),
1740199                    (uint32_t) proc_table[pid].address_data,
1740200                    (uint32_t) ((proc_table[pid].domain_data
1740201                                 +
1740202                                 proc_table[pid].extra_data)
1740203                                / 4096), 1, 0, 0);
1740204     }
1740205   else
1740206     {
1740207       gdt_segment (gdt_pid_to_segment_data (pid),
1740208                    (uint32_t) proc_table[pid].address_text,
1740209                    (uint32_t) ((proc_table[pid].domain_text
1740210                                 +
1740211                                 proc_table[pid].extra_data)
1740212                                / 4096), 1, 0, 0);
1740213     }
1740214   //
1740215   // ------------------------------------------------
1740216   // Might reload the GDT table, but it is not
1740217   // necessarily.
1740218   // Anyway, if you do it, nothing change. :-)
1740219   //
1740220   // gdt_load (&gdt_register);
1740221   // ------------------------------------------------
1740222   //
1740223   // Increase inode references for the working
1740224   // directory.
1740225   //
1740226   proc_table[pid].inode_cwd->references++;
1740227   //
1740228   // Duplicate valid file descriptors.
1740229   //
1740230   for (fdn = 0; fdn < OPEN_MAX; fdn++)
1740231     {
1740232       if (proc_table[ppid].fd[fdn].file != NULL
1740233           && (proc_table[ppid].fd[fdn].file->inode !=
1740234               NULL
1740235               || proc_table[ppid].fd[fdn].file->sock !=
1740236               NULL))
1740237         {
1740238           //
1740239           // Copy to the forked process.
```

```
1740240           //
1740241           proc_table[pid].fd[fdn].fl_flags
1740242             = proc_table[ppid].fd[fdn].fl_flags;
1740243           proc_table[pid].fd[fdn].fd_flags
1740244             = proc_table[ppid].fd[fdn].fd_flags;
1740245           proc_table[pid].fd[fdn].file =
1740246             proc_table[ppid].fd[fdn].file;
1740247           //
1740248           // Increment file reference.
1740249           //
1740250           proc_table[ppid].fd[fdn].file->references++;
1740251           //
1740252           // Check if it is a pipe and increment
1740253           // specific
1740254           // read/write reference counters inside the
1740255           // inode.
1740256           //
1740257           if (proc_table[ppid].fd[fdn].file->inode !=
1740258               NULL
1740259               && S_ISFIFO (proc_table[ppid].fd[fdn].
1740260                            file->inode->mode))
1740261             {
1740262               if (proc_table[ppid].fd[fdn].
1740263                   fl_flags & O_RDONLY)
1740264                 {
1740265                   proc_table[ppid].fd[fdn].file->
1740266                     inode->pipe_ref_read++;
1740267                 }
1740268               //
1740269               if (proc_table[ppid].fd[fdn].
1740270                   fl_flags & O_WRONLY)
1740271                 {
1740272                   proc_table[ppid].fd[fdn].file->
1740273                     inode->pipe_ref_write++;
1740274                 }
1740275             }
1740276         }
1740277     }
1740278   //
1740279   // Change segment descriptor values inside the
1740280   // stack,
1740281   // for: DS==ES==FS==GS.
1740282   //
1740283   // First calculate the absolute stack section
1740284   // address, from the
1740285   // kernel point of view.
1740286   //
1740287   if (allocated_data > 0)
1740288     {
1740289       addr_stack_pointer = allocated_data;
1740290     }
1740291   else
1740292     {
1740293       addr_stack_pointer = allocated_text;
1740294     }
1740295   //
1740296   // Then calculate the effective new stack pointer.
1740297   //
1740298   addr_stack_pointer += proc_table[pid].sp;
1740299   //
1740300   // Then calculate the segment descriptor, to be
1740301   // written
1740302   // inside the new process stack.
1740303   //
1740304   segment_descriptor = (gdt_pid_to_segment_data (pid) * 8);
1740305   //
1740306   // Then copy inside the stack the new values for
1740307   // data segments.
1740308   //
1740309   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1740310           (addr_stack_pointer + 28),
1740311           &segment_descriptor,
1740312           (sizeof segment_descriptor), NULL);
1740313   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1740314           (addr_stack_pointer + 32),
1740315           &segment_descriptor,
1740316           (sizeof segment_descriptor), NULL);
1740317   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1740318           (addr_stack_pointer + 36),
1740319           &segment_descriptor,
1740320           (sizeof segment_descriptor), NULL);
1740321   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1740322           (addr_stack_pointer + 40),
1740323           &segment_descriptor,
1740324           (sizeof segment_descriptor), NULL);
1740325   //
1740326   // Change segment descriptor value inside the stack
```

```
1740327   // for CS,
1740328   // if so is necessary.
1740329   //
1740330   segment_descriptor = (gdt_pid_to_segment_text (pid) * 8);
1740331   //
1740332   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
1740333           (addr_stack_pointer + 48),
1740334           &segment_descriptor,
1740335           (sizeof segment_descriptor), NULL);
1740336   //
1740337   // Set it ready.
1740338   //
1740339   proc_table[pid].status = PROC_READY;
1740340   //
1740341   // Return the new PID.
1740342   //
1740343   return (pid);
1740344 }
```

94.8.17 kernel/lib_s/s_fstat.c

«

Si veda la sezione 87.55.

```
1750001 #include <kernel/proc.h>
1750002 #include <kernel/lib_s.h>
1750003 #include <errno.h>
1750004 #include <fcntl.h>
1750005 //-----------------------------------------------------------
1750006 int
1750007 s_fstat (pid_t pid, int fdn, struct stat *buffer)
1750008 {
1750009   proc_t *ps;
1750010   inode_t *inode;
1750011   //
1750012   // Get process.
1750013   //
1750014   ps = proc_reference (pid);
1750015   //
1750016   // Verify if the file descriptor is valid.
1750017   //
1750018   if (ps->fd[fdn].file == NULL)
1750019     {
1750020       errset (EBADF);   // Bad file descriptor.
1750021       return (-1);
1750022     }
1750023   //
1750024   // Reach the inode.
1750025   //
1750026   inode = ps->fd[fdn].file->inode;
1750027   //
1750028   // If the Inode does not exist, exit with error.
1750029   //
1750030   if (inode == NULL)
1750031     {
1750032       errset (ENOENT);
1750033       return (-1);
1750034     }
1750035   //
1750036   // Inode loaded: update the buffer.
1750037   //
1750038   buffer->st_dev = inode->sb->device;
1750039   buffer->st_ino = inode->ino;
1750040   buffer->st_mode = inode->mode;
1750041   buffer->st_nlink = inode->links;
1750042   buffer->st_uid = inode->uid;
1750043   buffer->st_gid = inode->gid;
1750044   if (S_ISBLK (buffer->st_mode)
1750045       || S_ISCHR (buffer->st_mode))
1750046     {
1750047       buffer->st_rdev = inode->direct[0];
1750048     }
1750049   else
1750050     {
1750051       buffer->st_rdev = 0;
1750052     }
1750053   buffer->st_size = inode->size;
1750054   buffer->st_atime = inode->time;       // All times
1750055   // are the
1750056   // same for
1750057   buffer->st_mtime = inode->time;       // Minix 1
1750058   // file
1750059   // system.
1750060   buffer->st_ctime = inode->time;       //
1750061   buffer->st_blksize = inode->sb->blksize;
1750062   buffer->st_blocks = inode->blkcnt;
1750063   //
1750064   // If the inode is a device special file, the
```



```
1750065   // 'st_rdev' value is
1750066   // taken from the first direct zone (as of Minix 1
1750067   // organization).
1750068   //
1750069   if (S_ISBLK (inode->mode) || S_ISCHR (inode->mode))
1750070     {
1750071       buffer->st_rdev = inode->direct[0];
1750072     }
1750073   else
1750074     {
1750075       buffer->st_rdev = 0;
1750076     }
1750077   //
1750078   // Return.
1750079   //
1750080   return (0);
1750081 }
```

94.8.18 kernel/lib_s/s_ipconfig.c

«

Si veda la sezione 87.28.

```
1760001 #include <kernel/net.h>
1760002 #include <kernel/net/route.h>
1760003 #include <kernel/net/arp.h>
1760004 #include <errno.h>
1760005 #include <kernel/proc.h>
1760006 //-----------------------------------------------------------
1760007 // This syscall is present only inside os32.
1760008 //-----------------------------------------------------------
1760009 int
1760010 s_ipconfig (pid_t pid, int n, in_addr_t addr, int m)
1760011 {
1760012   //
1760013   // Must be a privileged process.
1760014   //
1760015   if (proc_table[pid].euid != 0)
1760016     {
1760017       errset (EPERM);
1760018       return (-1);
1760019     }
1760020   //
1760021   // Check arguments: net0 cannot be modified, because
1760022   // it is necessarily
1760023   // assigned to the loopback virtual interface.
1760024   //
1760025   if (n > NET_MAX_DEVICES || n < 1)
1760026     {
1760027       errset (EINVAL);
1760028       return (-1);
1760029     }
1760030   if (m > 32 || n < 0)
1760031     {
1760032       errset (EINVAL);
1760033       return (-1);
1760034     }
1760035   //
1760036   // Verify that the interface is present.
1760037   //
1760038   if (net_table[n].type == NET_DEV_NULL)
1760039     {
1760040       errset (ENODEV);
1760041       return (-1);
1760042     }
1760043   //
1760044   // Remove previous route related to the interface,
1760045   // if the interface
1760046   // was already configured.
1760047   //
1760048   if (net_table[n].ip != 0 && net_table[n].m != 0)
1760049     {
1760050       s_routedel (pid, net_table[n].ip, net_table[n].m);
1760051     }
1760052   //
1760053   // Modify the net_table[].
1760054   //
1760055   net_table[n].ip = ntohl (addr);
1760056   net_table[n].m = m;
1760057   //
1760058   // Add the route.
1760059   //
1760060   if (net_table[n].ip != 0 && net_table[n].m != 0)
1760061     {
1760062       return (s_routeadd (pid, addr, m, 0, n));
1760063     }
1760064   //
1760065   return (0);
```

```
}
```

94.8.19 kernel/lib_s/s_kill.c

Si veda la sezione 87.29.

```
#include <kernel/proc.h>
#include <kernel/lib_s.h>
#include <errno.h>
//-------------------------------------------------------
int
s_kill (pid_t pid_killer, pid_t pid_target, int sig)
{
  uid_t euid = proc_table[pid_killer].euid;
  uid_t uid = proc_table[pid_killer].uid;
  pid_t pgrp = proc_table[pid_killer].pgrp;
  int p;        // Index inside the process table.
  //
  if (pid_target < -1)
    {
      errset (ESRCH);
      return (-1);
    }
  else if (pid_target == -1)
    {
      if (sig == 0)
        {
          return (0);
        }
      if (euid == 0)
        {
          //
          // Because 'pid_target' is qual to '-1' and
          // the effective
          // user identity is '0', then, all
          // processes,
          // except the kernel and init, will receive
          // the signal.
          //
          // The following scan starts from 2, to
          // preserve the
          // kernel and init processes.
          //
          for (p = 2; p < PROCESS_MAX; p++)
            {
              if (proc_table[p].status != PROC_EMPTY
                  && proc_table[p].status != PROC_ZOMBIE)
                {
                  proc_sig_on (p, sig);
                }
            }
        }
      else
        {
          //
          // Because 'pid_target' is qual to '-1', but
          // the effective
          // user identity is not '0', then, all
          // processes owned
          // by the same effective user identity, will
          // receive the
          // signal.
          //
          // The following scan starts from 1, to
          // preserve the
          // kernel process.
          //
          for (p = 1; p < PROCESS_MAX; p++)
            {
              if (proc_table[p].status != PROC_EMPTY
                  && proc_table[p].status !=
                  PROC_ZOMBIE && proc_table[p].uid == euid)
                {
                  proc_sig_on (p, sig);
                }
            }
        }
      return (0);
    }
  else if (pid_target == 0)
    {
      if (sig == 0)
        {
          return (0);
        }
      //
      // The following scan starts from 1, to preserve
      // the
      // kernel process.
      //
      for (p = 1; p < PROCESS_MAX; p++)
        {
          if (proc_table[p].status != PROC_EMPTY
              && proc_table[p].status != PROC_ZOMBIE
              && proc_table[p].pgrp == pgrp)
            {
              proc_sig_on (p, sig);
            }
        }
      return (0);
    }
  else if (pid_target >= PROCESS_MAX)
    {
      errset (ESRCH);
      return (-1);
    }
  else  // (pid_target > 0)
    {
      if (proc_table[pid_target].status == PROC_EMPTY
          || proc_table[pid_target].status == PROC_ZOMBIE)
        {
          errset (ESRCH);
          return (-1);
        }
      else if (uid == proc_table[pid_target].uid ||
               uid == proc_table[pid_target].suid ||
               euid == proc_table[pid_target].uid ||
               euid == proc_table[pid_target].suid
               || euid == 0)
        {
          if (sig == 0)
            {
              return (0);
            }
          else
            {
              proc_sig_on (pid_target, sig);
              return (0);
            }
        }
      else
        {
          errset (EPERM);
          return (-1);
        }
    }
}
```

94.8.20 kernel/lib_s/s_link.c

Si veda la sezione 87.30.

```
#include <kernel/fs.h>
#include <errno.h>
#include <kernel/proc.h>
#include <kernel/lib_s.h>
//-------------------------------------------------------
int
s_link (pid_t pid, const char *path_old,
        const char *path_new)
{
  proc_t *ps;
  inode_t *inode_old;
  inode_t *inode_new;
  char path_new_full[PATH_MAX];
  //
  // Get process.
  //
  ps = proc_reference (pid);
  //
  // Try to get the old path inode.
  //
  inode_old = path_inode (pid, path_old);
  if (inode_old == NULL)
    {
      //
      // Cannot get the inode: 'errno' is already set
      // by
      // 'path_inode()'.
      //
      errset (errno);
      return (-1);
    }
  //
```

```
1780033   // The inode is available and checks are done:
1780034   // arrange to get a
1780035   // packed full path name and then the destination
1780036   // directory path.
1780037   //
1780038   path_full (path_new, ps->path_cwd, path_new_full);
1780039   //
1780040   //
1780041   //
1780042   inode_new =
1780043     path_inode_link (pid, path_new_full, inode_old,
1780044                      (mode_t) 0);
1780045   if (inode_new == NULL)
1780046     {
1780047       inode_put (inode_old);
1780048       return (-1);
1780049     }
1780050   if (inode_new != inode_old)
1780051     {
1780052       inode_put (inode_new);
1780053       inode_put (inode_old);
1780054       errset (EUNKNOWN);        // Unknown error.
1780055       return (-1);
1780056     }
1780057   //
1780058   // Inode data is already updated by
1780059   // 'path_inode_link()': just put
1780060   // it and return. Please note that only one is put,
1780061   // because it is
1780062   // just the same of the other.
1780063   //
1780064   inode_put (inode_new);
1780065   return (0);
1780066 }
```

94.8.21 kernel/lib_s/s_listen.c

Si veda la sezione 87.31.

```
1790001 #include <kernel/proc.h>
1790002 #include <kernel/lib_s.h>
1790003 #include <kernel/lib_k.h>
1790004 #include <errno.h>
1790005 #include <fcntl.h>
1790006 #include <sys/socket.h>
1790007 #include <arpa/inet.h>
1790008 //-----------------------------------------------------------
1790009 int
1790010 s_listen (pid_t pid, int sfdn, int backlog)
1790011 {
1790012   fd_t *sfd;
1790013   int s;
1790014   //
1790015   // Get file descriptor and verify that it is a
1790016   // socket.
1790017   //
1790018   sfd = fd_reference (pid, &sfdn);
1790019   if (sfd == NULL || sfd->file == NULL)
1790020     {
1790021       errset (EBADF);   // Bad file descriptor.
1790022       return (-1);
1790023     }
1790024   if (sfd->file->sock == NULL)
1790025     {
1790026       errset (ENOTSOCK);        // Not a socket.
1790027       return (-1);
1790028     }
1790029   if (sfd->file->sock->type != SOCK_STREAM)
1790030     {
1790031       errset (EOPNOTSUPP);      // Not a stream
1790032       // socket.
1790033       return (-1);
1790034     }
1790035   if (sfd->file->sock->raddr != 0
1790036       || sfd->file->sock->rport != 0)
1790037     {
1790038       //
1790039       // The socket is connected, and cannot be good
1790040       // for
1790041       // listening.
1790042       //
1790043       errset (EISCONN);
1790044       return (-1);
1790045     }
1790046   //
1790047   // Scan the other sockets to find if there is
1790048   // another one listening.
```



```
1790049   //
1790050   for (s = 0; s < SOCK_MAX_SLOTS; s++)
1790051     {
1790052       if (sock_table[s].tcp.conn == TCP_LISTEN
1790053           && sock_table[s].lport == sfd->file->sock->lport)
1790054         {
1790055           //
1790056           // Yes, there is one: sorry.
1790057           //
1790058           errset (EADDRINUSE);
1790059           return (-1);
1790060         }
1790061     }
1790062   //
1790063   // Check the current TCP state.
1790064   //
1790065   if (sfd->file->sock->tcp.conn != 0
1790066       && sfd->file->sock->tcp.conn != TCP_LISTEN)
1790067     {
1790068       //
1790069       // Cannot change the socket stream state.
1790070       //
1790071       errset (EISCONN);
1790072       return (-1);
1790073     }
1790074   //
1790075   // The socket might be already listening, but the
1790076   // newly requested
1790077   // queue should be greater or equal to the previous
1790078   // one.
1790079   //
1790080   if (sfd->file->sock->tcp.conn == TCP_LISTEN
1790081       && backlog < sfd->file->sock->tcp.listen_max)
1790082     {
1790083       //
1790084       // Cannot reduce the listen queue: just ignore.
1790085       //
1790086       return (0);
1790087     }
1790088   //
1790089   // Ok.
1790090   //
1790091   sfd->file->sock->tcp.conn = TCP_LISTEN;
1790092   sfd->file->sock->tcp.listen_max =
1790093     min (backlog, SOCK_MAX_QUEUE);
1790094   sfd->file->sock->tcp.listen_pid = pid;
1790095   return (0);
1790096 }
```

94.8.22 kernel/lib_s/s_longjmp.c

Si veda la sezione 87.49.

```
1800001 #include <kernel/lib_s.h>
1800002 #include <kernel/proc.h>
1800003 #include <errno.h>
1800004 #include <sys/os32.h>
1800005 //-----------------------------------------------------------
1800006 extern uint32_t proc_stack_pointer;
1800007 //-----------------------------------------------------------
1800008 void
1800009 s_longjmp (pid_t pid, jmp_buf env, int val)
1800010 {
1800011   jmp_stack_t *sp;
1800012   jmp_env_t *jmpenv;
1800013   //
1800014   // Translate the pointer 'env', to the kernel point
1800015   // of view.
1800016   //
1800017   jmpenv = ptr (pid, env);
1800018   //
1800019   // Find where *was* the process stack in memory,
1800020   // from the kernel point
1800021   // of view. Please notice that the current stack at
1800022   // 'proc_stack_pointer' will be saved from the
1800023   // scheduler inside
1800024   // the process table. So, the replacement is made at
1800025   // the current
1800026   // stack position, and not inside the process table.
1800027   //
1800028   sp = ptr (pid, (void *) jmpenv->esp0);
1800029   //
1800030   // Restore the process stack.
1800031   //
1800032   sp->eax0 = jmpenv->eax0;
1800033   sp->ecx0 = jmpenv->ecx0;
1800034   sp->edx0 = jmpenv->edx0;
```

```
1800035    sp->ebx0 = jmpenv->ebx0;
1800036    sp->ebp0 = jmpenv->ebp0;
1800037    sp->esi0 = jmpenv->esi0;
1800038    sp->edi0 = jmpenv->edi0;
1800039    sp->ds0 = jmpenv->ds0;
1800040    sp->es0 = jmpenv->es0;
1800041    sp->fs0 = jmpenv->fs0;
1800042    sp->gs0 = jmpenv->gs0;
1800043    sp->eflags0 = jmpenv->eflags0;
1800044    sp->cs0 = jmpenv->cs0;
1800045    sp->eip0 = jmpenv->eip0;
1800046    //
1800047    sp->eip1 = jmpenv->eip1;
1800048    sp->syscallnr = jmpenv->syscallnr;
1800049    sp->msg_pointer = jmpenv->msg_pointer;
1800050    sp->msg_size = jmpenv->msg_size;
1800051    sp->env = jmpenv->env;
1800052    sp->ret = val;
1800053    sp->ebp1 = jmpenv->ebp1;
1800054    sp->eip2 = jmpenv->eip2;
1800055    //
1800056    // Replace the stack pointer too!
1800057    //
1800058    proc_stack_pointer = jmpenv->esp0;
1800059  }
```

94.8.23 kernel/lib_s/s_lseek.c

Si veda la sezione 87.33.

```
1810001  #include <kernel/proc.h>
1810002  #include <kernel/lib_s.h>
1810003  #include <errno.h>
1810004  //-------------------------------------------------------
1810005  off_t
1810006  s_lseek (pid_t pid, int fdn, off_t offset, int whence)
1810007  {
1810008    inode_t *inode;
1810009    file_t *file;
1810010    fd_t *fd;
1810011    off_t test_offset;
1810012    //
1810013    // Get file descriptor.
1810014    //
1810015    fd = fd_reference (pid, &fdn);
1810016    if (fd == NULL || fd->file == NULL
1810017        || fd->file->inode == NULL)
1810018      {
1810019        errset (EBADF);   // Bad file descriptor.
1810020        return (-1);
1810021      }
1810022    //
1810023    // Get file table item.
1810024    //
1810025    file = fd->file;
1810026    //
1810027    // Get inode.
1810028    //
1810029    inode = file->inode;
1810030    //
1810031    // Change position depending on the 'whence'
1810032    // parameter.
1810033    //
1810034    if (whence == SEEK_SET)
1810035      {
1810036        if (offset < 0)
1810037          {
1810038            errset (EINVAL);      // Invalid argument.
1810039            return ((off_t) - 1);
1810040          }
1810041        else
1810042          {
1810043            fd->file->offset = offset;
1810044          }
1810045      }
1810046    else if (whence == SEEK_CUR)
1810047      {
1810048        test_offset = fd->file->offset;
1810049        test_offset += offset;
1810050        if (test_offset < 0)
1810051          {
1810052            errset (EINVAL);      // Invalid argument.
1810053            return ((off_t) - 1);
1810054          }
1810055        else
1810056          {
1810057            fd->file->offset = test_offset;
1810058          }
1810059      }
1810060    else if (whence == SEEK_END)
1810061      {
1810062        test_offset = inode->size;
1810063        test_offset += offset;
1810064        if (test_offset < 0)
1810065          {
1810066            errset (EINVAL);      // Invalid argument.
1810067            return ((off_t) - 1);
1810068          }
1810069        else
1810070          {
1810071            fd->file->offset = test_offset;
1810072          }
1810073      }
1810074    else
1810075      {
1810076        errset (EINVAL);  // Invalid argument.
1810077        return ((off_t) - 1);
1810078      }
1810079    //
1810080    // Return the new file position.
1810081    //
1810082    return (fd->file->offset);
1810083  }
```

94.8.24 kernel/lib_s/s_mkdir.c

Si veda la sezione 87.34.

```
1820001  #include <kernel/fs.h>
1820002  #include <errno.h>
1820003  #include <kernel/proc.h>
1820004  #include <libgen.h>
1820005  #include <kernel/lib_s.h>
1820006  //-------------------------------------------------------
1820007  int
1820008  s_mkdir (pid_t pid, const char *path, mode_t mode)
1820009  {
1820010    proc_t *ps;
1820011    inode_t *inode_directory;
1820012    inode_t *inode_parent;
1820013    int status;
1820014    char path_directory[PATH_MAX];
1820015    char path_copy[PATH_MAX];
1820016    char *path_parent;
1820017    ssize_t size_written;
1820018    //
1820019    struct
1820020    {
1820021      ino_t ino_1;
1820022      char name_1[NAME_MAX];
1820023      ino_t ino_2;
1820024      char name_2[NAME_MAX];
1820025    } directory;
1820026    //
1820027    // Get process.
1820028    //
1820029    ps = proc_reference (pid);
1820030    //
1820031    // Correct the mode with the umask.
1820032    //
1820033    mode &= ~ps->umask;
1820034    //
1820035    // Inside 'mode', the file type is fixed. No check
1820036    // is made.
1820037    //
1820038    mode &= 00777;
1820039    mode |= S_IFDIR;
1820040    //
1820041    // The full path and the directory path is needed.
1820042    //
1820043    status = path_full (path, ps->path_cwd, path_directory);
1820044    if (status < 0)
1820045      {
1820046        return (-1);
1820047      }
1820048    strncpy (path_copy, path_directory, PATH_MAX);
1820049    path_copy[PATH_MAX - 1] = 0;
1820050    path_parent = dirname (path_copy);
1820051    //
1820052    // Check if something already exists with the same
1820053    // name. The scan
1820054    // is done with kernel privileges.
1820055    //
1820056    inode_directory = path_inode ((uid_t) 0, path_directory);
```

```
1820057   if (inode_directory != NULL)
1820058     {
1820059       //
1820060       // The file already exists. Put inode and return
1820061       // an error.
1820062       //
1820063       inode_put (inode_directory);
1820064       errset (EEXIST);  // File exists.
1820065       return (-1);
1820066     }
1820067   //
1820068   // Try to locate the directory that should contain
1820069   // this one.
1820070   //
1820071   inode_parent = path_inode (pid, path_parent);
1820072   if (inode_parent == NULL)
1820073     {
1820074       //
1820075       // Cannot locate the directory: return an error.
1820076       // The variable
1820077       // 'errno' should already be set by
1820078       // 'path_inode()'.
1820079       //
1820080       errset (errno);
1820081       return (-1);
1820082     }
1820083   //
1820084   // Try to create the node: should fail if the user
1820085   // does not have
1820086   // enough permissions.
1820087   //
1820088   inode_directory =
1820089     path_inode_link (pid, path_directory, NULL, mode);
1820090   if (inode_directory == NULL)
1820091     {
1820092       //
1820093       // Sorry: cannot create the inode! The variable
1820094       // 'errno' should
1820095       // already be set by 'path_inode_link()'.
1820096       //
1820097       errset (errno);
1820098       return (-1);
1820099     }
1820100   //
1820101   // Fill records for '.' and '..'.
1820102   //
1820103   directory.ino_1 = inode_directory->ino;
1820104   strncpy (directory.name_1, ".", (size_t) 3);
1820105   directory.ino_2 = inode_parent->ino;
1820106   strncpy (directory.name_2, "..", (size_t) 3);
1820107   //
1820108   // Write data.
1820109   //
1820110   size_written =
1820111     inode_file_write (inode_directory, (off_t) 0,
1820112                       &directory, (sizeof directory));
1820113   if (size_written != (sizeof directory))
1820114     {
1820115       return (-1);
1820116     }
1820117   //
1820118   // Fix directory inode links.
1820119   //
1820120   inode_directory->links = 2;
1820121   inode_directory->time = s_time (pid, NULL);
1820122   inode_directory->changed = 1;
1820123   //
1820124   // Fix parent directory inode links.
1820125   //
1820126   inode_parent->links++;
1820127   inode_parent->time = s_time (pid, NULL);
1820128   inode_parent->changed = 1;
1820129   //
1820130   // Save and put the inodes.
1820131   //
1820132   inode_save (inode_parent);
1820133   inode_save (inode_directory);
1820134   inode_put (inode_parent);
1820135   inode_put (inode_directory);
1820136   //
1820137   // Return.
1820138   //
1820139   return (0);
1820140 }
```


94.8.25 kernel/lib_s/s_mknod.c

«

Si veda la sezione 87.35.

```
1830001 #include <kernel/fs.h>
1830002 #include <errno.h>
1830003 #include <kernel/proc.h>
1830004 #include <kernel/lib_s.h>
1830005 //-------------------------------------------------------
1830006 int
1830007 s_mknod (pid_t pid, const char *path, mode_t mode,
1830008          dev_t device)
1830009 {
1830010   proc_t *ps;
1830011   inode_t *inode;
1830012   char full_path[PATH_MAX];
1830013   //
1830014   // Get process.
1830015   //
1830016   ps = proc_reference (pid);
1830017   //
1830018   // Correct the mode with the umask.
1830019   //
1830020   mode &= ~ps->umask;
1830021   //
1830022   // Currently must be root, unless the type is a
1830023   // regular file,
1830024   // or a FIFO file.
1830025   //
1830026   if (!(S_ISFIFO (mode) || S_ISREG (mode)))
1830027     {
1830028       if (ps->uid != 0)
1830029         {
1830030           errset (EPERM);       // Operation not
1830031           // permitted.
1830032           return (-1);
1830033         }
1830034     }
1830035   //
1830036   // Check the type of node requested.
1830037   //
1830038   if (!(S_ISBLK (mode) ||
1830039         S_ISCHR (mode) ||
1830040         S_ISREG (mode) || S_ISFIFO (mode)
1830041         || S_ISDIR (mode)))
1830042     {
1830043       errset (EINVAL);  // Invalid argument.
1830044       return (-1);
1830045     }
1830046   //
1830047   // Check if something already exists with the same
1830048   // name.
1830049   //
1830050   inode = path_inode (pid, path);
1830051   if (inode != NULL)
1830052     {
1830053       //
1830054       // The file already exists. Put inode and return
1830055       // an error.
1830056       //
1830057       inode_put (inode);
1830058       errset (EEXIST);  // File exists.
1830059       return (-1);
1830060     }
1830061   //
1830062   // Try to creat the node.
1830063   //
1830064   path_full (path, ps->path_cwd, full_path);
1830065   inode = path_inode_link (pid, full_path, NULL, mode);
1830066   if (inode == NULL)
1830067     {
1830068       //
1830069       // Sorry: cannot create the inode!
1830070       //
1830071       return (-1);
1830072     }
1830073   //
1830074   // Set the device number if necessary.
1830075   //
1830076   if (S_ISBLK (mode) || S_ISCHR (mode))
1830077     {
1830078       inode->direct[0] = device;
1830079       inode->changed = 1;
1830080     }
1830081   //
1830082   // Put the inode.
1830083   //
1830084   inode_put (inode);
1830085   //
```

```
1830086   // Return.
1830087   //
1830088   return (0);
1830089 }
```

94.8.26 kernel/lib_s/s_mount.c

«

Si veda la sezione 87.36.

```
1840001 #include <kernel/fs.h>
1840002 #include <errno.h>
1840003 #include <kernel/proc.h>
1840004 #include <kernel/lib_s.h>
1840005 //-----------------------------------------------------------
1840006 int
1840007 s_mount (pid_t pid, const char *path_dev,
1840008          const char *path_mnt, int options)
1840009 {
1840010   proc_t *ps;
1840011   dev_t device; // Device to mount.
1840012   inode_t *inode_mnt;   // Directory mount point.
1840013   void *pstatus;
1840014   //
1840015   // Get process.
1840016   //
1840017   ps = proc_reference (pid);
1840018   //
1840019   // Verify to be the super user.
1840020   //
1840021   if (ps->euid != 0)
1840022     {
1840023       errset (EPERM);   // Operation not permitted.
1840024       return (-1);
1840025     }
1840026   //
1840027   device = path_device (pid, path_dev);
1840028   if (device < 0)
1840029     {
1840030       return (-1);
1840031     }
1840032   //
1840033   inode_mnt = path_inode (pid, path_mnt);
1840034   if (inode_mnt == NULL)
1840035     {
1840036       return (-1);
1840037     }
1840038   if (!S_ISDIR (inode_mnt->mode))
1840039     {
1840040       inode_put (inode_mnt);
1840041       errset (ENOTDIR); // Not a directory.
1840042       return (-1);
1840043     }
1840044   if (inode_mnt->sb_attached != NULL)
1840045     {
1840046       inode_put (inode_mnt);
1840047       errset (EBUSY);   // Device or resource busy.
1840048       return (-1);
1840049     }
1840050   //
1840051   // All data is available.
1840052   //
1840053   pstatus = sb_mount (device, &inode_mnt, options);
1840054   if (pstatus == NULL)
1840055     {
1840056       inode_put (inode_mnt);
1840057       return (-1);
1840058     }
1840059   //
1840060   return (0);
1840061 }
```

94.8.27 kernel/lib_s/s_open.c

«

Si veda la sezione 87.37.

```
1850001 #include <kernel/proc.h>
1850002 #include <kernel/lib_s.h>
1850003 #include <kernel/lib_k.h>
1850004 #include <errno.h>
1850005 #include <fcntl.h>
1850006 //-----------------------------------------------------------
1850007 int
1850008 s_open (pid_t pid, const char *path, int oflags,
1850009         mode_t mode)
1850010 {
1850011   inode_t *inode;
1850012   int status;
```



```
1850013   file_t *file;
1850014   fd_t *fd;
1850015   int fdn;
1850016   char full_path[PATH_MAX];
1850017   int perm;
1850018   tty_t *tty;
1850019   mode_t umask;
1850020   int errno_save;
1850021   //
1850022   // k_printf ("%s(%i, %s, %x, %05o)\n", __func__,
1850023   // (int) pid,
1850024   // path, oflags, (int) mode);
1850025   //
1850026   // Check path argument.
1850027   //
1850028   if (path == NULL || strlen (path) == 0)
1850029     {
1850030       errset (EINVAL);  // Invalid argument.
1850031       return (-1);
1850032     }
1850033   //
1850034   // Correct the mode with the umask. As it is not a
1850035   // directory, to the
1850036   // mode are removed execution and sticky
1850037   // permissions.
1850038   //
1850039   umask = proc_table[pid].umask | 01111;
1850040   mode &= ~umask;
1850041   //
1850042   // Check open options.
1850043   //
1850044   if (oflags & O_WRONLY)
1850045     {
1850046       //
1850047       // The file is to be opened for write, or for
1850048       // read/write.
1850049       // Try to get inode.
1850050       //
1850051       inode = path_inode (pid, path);
1850052       if (inode == NULL)
1850053         {
1850054           //
1850055           // Cannot get the inode. See if there is the
1850056           // creation
1850057           // option.
1850058           //
1850059           if (oflags & O_CREAT)
1850060             {
1850061               //
1850062               // Try to create the missing inode: the
1850063               // file must be a
1850064               // regular one, so add the mode.
1850065               //
1850066               path_full (path,
1850067                          proc_table[pid].path_cwd,
1850068                          full_path);
1850069               inode =
1850070                 path_inode_link (pid, full_path, NULL,
1850071                                  (mode | S_IFREG));
1850072               if (inode == NULL)
1850073                 {
1850074                   //
1850075                   // Sorry: cannot create the inode!
1850076                   // Variable ‘errno’
1850077                   // is already set by
1850078                   // ‘path_inode_link()’.
1850079                   //
1850080                   errset (errno);
1850081                   return (-1);
1850082                 }
1850083             }
1850084           else
1850085             {
1850086               //
1850087               // Cannot open the inode. Variable
1850088               // ‘errno’
1850089               // should be already set by
1850090               // ‘path_inode()’.
1850091               //
1850092               errset (errno);
1850093               return (-1);
1850094             }
1850095         }
1850096       //
1850097       // The inode was read or created: check if it
1850098       // must be
1850099       // truncated. It can be truncated only if it is
```

```
      // a regular
      // file.
      //
      if (oflags & O_TRUNC && inode->mode & S_IFREG)
        {
          //
          // Truncate inode.
          //
          status = inode_truncate (inode);
          if (status != 0)
            {
              //
              // Cannot truncate the inode: release it
              // and return.
              // But this error should never happen,
              // because the
              // function 'inode_truncate()' will not
              // return any
              // other value than zero.
              //
              errno_save = errno;
              inode_put (inode);
              errset (errno_save);
              return (-1);
            }
        }
    }
  else
    {
      //
      // The file is to be opened for read, but not
      // for write.
      // Try to get inode.
      //
      inode = path_inode (pid, path);
      if (inode == NULL)
        {
          //
          // Cannot open the file.
          //
          errset (errno);
          return (-1);
        }
    }
  //
  // An inode was opened: check type and access
  // permissions.
  // All file types are good, even directories, as the
  // type
  // DIR is implemented through file descriptors.
  //
  perm = 0;
  if (oflags & O_RDONLY)
    perm |= 4;
  if (oflags & O_WRONLY)
    perm |= 2;
  status =
    inode_check (inode, S_IFMT, perm,
                 proc_table[pid].euid,
                 proc_table[pid].egid);
  if (status != 0)
    {
      //
      // The file type is not correct or the user does
      // not have
      // permissions.
      //
      return (-1);
    }
  //
  // Allocate the file, inside the file table.
  //
  file = file_reference (-1);
  if (file == NULL)
    {
      //
      // Cannot allocate the file inside the file
      // table: release the
      // inode, update 'errno' and return.
      //
      inode_put (inode);
      errset (ENFILE);  // Too many files open in
      // system.
      return (-1);
    }
  //
  // Put some data inside the file item. Only options
```

```
  // O_RDONLY and O_WRONLY are kept here, because the
  // O_APPEND
  // is saved inside the file descriptor table.
  //
  file->references = 1;
  file->oflags = (oflags & (O_RDONLY | O_WRONLY));
  file->inode = inode;
  file->sock = NULL;
  //
  // Allocate the file descriptor: variable 'fdn' will
  // be modified
  // by the call to 'fd_reference()'.
  //
  fdn = -1;
  fd = fd_reference (pid, &fdn);
  if (fd == NULL)
    {
      //
      // Cannot allocate the file descriptor: remove
      // the item from
      // file table.
      //
      file->references = 0;
      file->oflags = 0;
      file->inode = NULL;
      file->sock = NULL;
      //
      // Release the inode.
      //
      inode_put (inode);
      //
      // Return an error.
      //
      errset (EMFILE);  // Too many open files.
      return (-1);
    }
  //
  // File descriptor allocated: put some data inside
  // the
  // file descriptor item.
  //
  fd->fl_flags =
    (oflags & (O_RDONLY | O_WRONLY | O_APPEND));
  fd->fd_flags = 0;
  fd->file = file;
  fd->file->offset = 0;
  //
  // Check for particular types and situations.
  //
  if ((S_ISCHR (inode->mode))
      && (oflags & O_RDONLY) && (oflags & O_WRONLY))
    {
      //
      // The inode is a character special file
      // (related to a character
      // device), opened for read and write!
      //
      if ((inode->direct[0] & 0xFF00) ==
          (DEV_CONSOLE_MAJOR << 8))
        {
          //
          // It is a terminal (currently only consoles
          // are possible).
          // Get the tty reference.
          //
          tty = tty_reference ((dev_t) inode->direct[0]);
          //
          // Verify that the terminal is not already
          // the controlling
          // terminal of some process group.
          //
          if (tty->pgrp == 0)
            {
              //
              // The terminal is free: verify if the
              // current process
              // needs a controlling terminal.
              //
              if (proc_table[pid].device_tty == 0
                  && proc_table[pid].pgrp == pid)
                {
                  //
                  // It is a group leader with no
                  // controlling
                  // terminal: set the controlling
                  // terminal.
                  //
```

```
1850274                   proc_table[pid].device_tty =
1850275                     inode->direct[0];
1850276                   tty->pgrp = proc_table[pid].pgrp;
1850277                 }
1850278             }
1850279         }
1850280     }
1850281   else if (S_ISFIFO (inode->mode))
1850282     {
1850283       //
1850284       // It is FIFO (named pipe).
1850285       //
1850286       if ((oflags & O_ACCMODE) == O_RDWR)
1850287         {
1850288           inode->pipe_ref_read++;
1850289           inode->pipe_ref_write++;
1850290         }
1850291       else if (oflags & O_RDONLY)
1850292         {
1850293           inode->pipe_ref_read++;
1850294           //
1850295           // Go to sleep if there are no processes
1850296           // writing to the
1850297           // inode. Otherwise, wake them up.
1850298           //
1850299           if (inode->pipe_ref_write == 0)
1850300             {
1850301               proc_table[pid].status = PROC_SLEEPING;
1850302               proc_table[pid].ret = 0;
1850303               proc_table[pid].wakeup_inode = inode;
1850304               proc_table[pid].wakeup_events =
1850305                 WAKEUP_EVENT_PIPE_READ;
1850306             }
1850307           else
1850308             {
1850309               proc_wakeup_pipe_write (inode);
1850310             }
1850311         }
1850312       else if (oflags & O_WRONLY)
1850313         {
1850314           inode->pipe_ref_write++;
1850315           //
1850316           // Go to sleep if there are no processes
1850317           // reading to the
1850318           // inode. Otherwise, wake them up.
1850319           //
1850320           if (inode->pipe_ref_read == 0)
1850321             {
1850322               proc_table[pid].status = PROC_SLEEPING;
1850323               proc_table[pid].ret = 0;
1850324               proc_table[pid].wakeup_inode = inode;
1850325               proc_table[pid].wakeup_events =
1850326                 WAKEUP_EVENT_PIPE_WRITE;
1850327             }
1850328           else
1850329             {
1850330               proc_wakeup_pipe_read (inode);
1850331             }
1850332         }
1850333     }
1850334   //
1850335   // Return the file descriptor.
1850336   //
1850337   return (fdn);
1850338 }
```

94.8.28 kernel/lib_s/s_pipe.c

«

Si veda la sezione 87.38.

```
1860001 #include <kernel/proc.h>
1860002 #include <kernel/lib_s.h>
1860003 #include <kernel/lib_k.h>
1860004 #include <errno.h>
1860005 #include <fcntl.h>
1860006 //-----------------------------------------------------------
1860007 int
1860008 s_pipe (pid_t pid, int pipefd[2])
1860009 {
1860010   file_t *file;
1860011   fd_t *fd_read;
1860012   fd_t *fd_write;
1860013   int fdn_read;
1860014   int fdn_write;
1860015   //
1860016   // Allocate the file inside the file table and the
1860017   // inode inside
1860018   // the inode table.
1860019   //
1860020   file = file_pipe_make ();
1860021   if (file == NULL)
1860022     {
1860023       errset (errno);
1860024       return (-1);
1860025     }
1860026   //
1860027   // Prepare file descriptor for read.
1860028   //
1860029   fdn_read = -1;
1860030   fd_read = fd_reference (pid, &fdn_read);
1860031   if (fd_read == NULL)
1860032     {
1860033       //
1860034       // Cannot allocate the file descriptor: remove
1860035       // the item from
1860036       // file table and put the relative inode.
1860037       //
1860038       file->references = 0;
1860039       file->oflags = 0;
1860040       inode_put (file->inode);
1860041       file->inode = NULL;
1860042       //
1860043       // Return an error.
1860044       //
1860045       errset (EMFILE);  // Too many open files.
1860046       return (-1);
1860047     }
1860048   //
1860049   // File descriptor allocated: put some data inside
1860050   // the
1860051   // file descriptor item and increment the pipe
1860052   // references
1860053   // for read.
1860054   //
1860055   fd_read->fl_flags = O_RDONLY;
1860056   fd_read->fd_flags = 0;
1860057   fd_read->file = file;
1860058   fd_read->file->offset = 0;
1860059   fd_read->file->inode->pipe_ref_read++;
1860060   //
1860061   // Prepare file descriptor for write.
1860062   //
1860063   fdn_write = -1;
1860064   fd_write = fd_reference (pid, &fdn_write);
1860065   if (fd_write == NULL)
1860066     {
1860067       //
1860068       // Cannot allocate the file descriptor: remove
1860069       // the item from
1860070       // file table and put the relative inode.
1860071       //
1860072       file->references = 0;
1860073       file->oflags = 0;
1860074       inode_put (file->inode);
1860075       file->inode = NULL;
1860076       //
1860077       // Remove file descriptor for read.
1860078       //
1860079       fd_read->file->inode->pipe_ref_read--;
1860080       fd_read->fl_flags = 0;
1860081       fd_read->fd_flags = 0;
1860082       fd_read->file = NULL;
1860083       //
1860084       // Return an error.
1860085       //
1860086       errset (EMFILE);  // Too many open files.
1860087       return (-1);
1860088     }
1860089   //
1860090   // File descriptor allocated: put some data inside
1860091   // the
1860092   // file descriptor item.
1860093   //
1860094   fd_write->fl_flags = O_WRONLY;
1860095   fd_write->fd_flags = 0;
1860096   fd_write->file = file;
1860097   fd_write->file->offset = 0;
1860098   fd_write->file->inode->pipe_ref_write++;
1860099   //
1860100   // Save file descriptor numbers inside the
1860101   // 'pipefd[]' array.
1860102   //
1860103   pipefd[0] = fdn_read;
1860104   pipefd[1] = fdn_write;
```

```
1860105     //
1860106     // Ok.
1860107     //
1860108     return (0);
1860109   }
```

94.8.29 kernel/lib_s/s_read.c

«

Si veda la sezione 87.39.

```
1870001 #include <kernel/proc.h>
1870002 #include <kernel/lib_s.h>
1870003 #include <errno.h>
1870004 #include <fcntl.h>
1870005 //-----------------------------------------------------------
1870006 #define DEBUG 0
1870007 //-----------------------------------------------------------
1870008 ssize_t
1870009 s_read (pid_t pid, int fdn, void *buffer, size_t count)
1870010 {
1870011   fd_t *fd;
1870012   ssize_t size_read;
1870013   int eof = 0;
1870014   //
1870015   // Get file descriptor.
1870016   //
1870017   fd = fd_reference (pid, &fdn);
1870018   if (fd == NULL || fd->file == NULL
1870019       || (fd->file->inode == NULL
1870020           && fd->file->sock == NULL))
1870021     {
1870022       errset (EBADF);   // Bad file descriptor.
1870023       return ((ssize_t) - 1);
1870024     }
1870025   //
1870026   // Check if it is opened for read.
1870027   //
1870028   if (!(fd->file->oflags & O_RDONLY))
1870029     {
1870030       //
1870031       // The file is not opened for read.
1870032       //
1870033       errset (EINVAL);  // Invalid argument.
1870034       return ((ssize_t) - 1);
1870035     }
1870036   //
1870037   // Check the kind of file to be read and read it.
1870038   //
1870039   if (fd->file->sock != NULL)
1870040     {
1870041       //
1870042       // Read from the socket and return.
1870043       //
1870044       return (s_recvfrom
1870045               (pid, fdn, buffer, count, 0, NULL, NULL));
1870046     }
1870047   else if (S_ISBLK (fd->file->inode->mode)
1870048            || S_ISCHR (fd->file->inode->mode))
1870049     {
1870050       //
1870051       // A device is to be read.
1870052       //
1870053       size_read =
1870054         dev_io (pid,
1870055                 (dev_t) fd->file->inode->direct[0],
1870056                 DEV_READ, fd->file->offset, buffer,
1870057                 count, &eof);
1870058       if (size_read < 0
1870059           && (errno == EAGAIN || errno == EWOULDBLOCK))
1870060         {
1870061           if (fd->fl_flags & O_NONBLOCK)
1870062             {
1870063               //
1870064               // Non blocking null read.
1870065               //
1870066               ;
1870067             }
1870068           else
1870069             {
1870070               //
1870071               // Null read: put the process to sleep.
1870072               //
1870073               proc_table[pid].status = PROC_SLEEPING;
1870074               proc_table[pid].ret = 0;
1870075               proc_table[pid].wakeup_events =
1870076                 WAKEUP_EVENT_DEV_READ;
1870077               proc_table[pid].wakeup_dev =
1870078                 fd->file->inode->direct[0];
1870079               if (DEBUG)
1870080                 {
1870081                   k_printf
1870082                     ("[%s] PID %i goes to sleep "
1870083                      "waiting to read from a "
1870084                      "device.\n", __FILE__, pid);
1870085                 }
1870086             }
1870087         }
1870088     }
1870089   else if (S_ISREG (fd->file->inode->mode))
1870090     {
1870091       //
1870092       // A regular file is to be read.
1870093       //
1870094       size_read =
1870095         inode_file_read (fd->file->inode,
1870096                          fd->file->offset, buffer,
1870097                          count, &eof);
1870098     }
1870099   else if (S_ISDIR (fd->file->inode->mode))
1870100     {
1870101       //
1870102       // A directory, is to be read.
1870103       //
1870104       size_read =
1870105         inode_file_read (fd->file->inode,
1870106                          fd->file->offset, buffer,
1870107                          count, &eof);
1870108     }
1870109   else if (S_ISFIFO (fd->file->inode->mode))
1870110     {
1870111       //
1870112       // A pipe, is to be read.
1870113       //
1870114       size_read =
1870115         inode_pipe_read (fd->file->inode, buffer,
1870116                          count, &eof);
1870117       //
1870118       if (size_read == 0)
1870119         {
1870120           //
1870121           // Check what to do.
1870122           //
1870123           if (fd->file->inode->pipe_ref_write == 0)
1870124             {
1870125               //
1870126               // EOF, if it is a valid pointer, is
1870127               // already
1870128               // set by 'inode_pipe_read()', if is
1870129               // time to
1870130               // set it.
1870131               //
1870132               // Wake up processes waiting to write.
1870133               //
1870134               proc_wakeup_pipe_write (fd->file->inode);
1870135               //
1870136               return (size_read);
1870137             }
1870138           else
1870139             {
1870140               //
1870141               // Go to sleep.
1870142               //
1870143               proc_table[pid].status = PROC_SLEEPING;
1870144               proc_table[pid].ret = 0;
1870145               proc_table[pid].wakeup_inode =
1870146                 fd->file->inode;
1870147               proc_table[pid].wakeup_events =
1870148                 WAKEUP_EVENT_PIPE_READ;
1870149               if (DEBUG)
1870150                 {
1870151                   k_printf
1870152                     ("[%s] PID %i goes to sleep "
1870153                      "waiting to read from a pipe.\n",
1870154                      __FILE__, pid);
1870155                 }
1870156             }
1870157         }
1870158       else
1870159         {
1870160           //
1870161           // Wake up processes waiting to write.
1870162           //
1870163           proc_wakeup_pipe_write (fd->file->inode);
1870164         }
```

```
1870165       }
1870166     else
1870167       {
1870168         //
1870169         // Unsupported file type.
1870170         //
1870171         errset (E_FILE_TYPE_UNSUPPORTED); // File type
1870172         // unsupported.
1870173         return ((ssize_t) - 1);
1870174       }
1870175     //
1870176     // Update the file descriptor internal offset, if
1870177     // there is an inode.
1870178     //
1870179     if (fd->file->inode != NULL && size_read > 0)
1870180       {
1870181         fd->file->offset += size_read;
1870182       }
1870183     //
1870184     // Return the size read, even if it is an error.
1870185     // Please notice
1870186     // that a size of zero might be related to an end of
1870187     // file, or
1870188     // just that the read should be retried. For the
1870189     // latter case,
1870190     // -1 is returned with error EAGAIN, so that the
1870191     // function
1870192     // 'read()' can retry.
1870193     //
1870194     if (size_read == 0 && !eof)
1870195       {
1870196         errset (EAGAIN);
1870197         return (-1);
1870198       }
1870199     else
1870200       {
1870201         return (size_read);
1870202       }
1870203 }
```

94.8.30 kernel/lib_s/s_recvfrom.c

«

Si veda la sezione 87.40.

```
1880001 #include <fcntl.h>
1880002 #include <kernel/proc.h>
1880003 #include <kernel/net.h>
1880004 #include <kernel/net/route.h>
1880005 #include <errno.h>
1880006 #include <arpa/inet.h>
1880007 #include <sys/os32.h>
1880008 #include <netinet/udp.h>
1880009 //-------------------------------------------------------
1880010 #define DEBUG 0
1880011 //-------------------------------------------------------
1880012 ssize_t
1880013 s_recvfrom (pid_t pid, int sfdn, void *buffer,
1880014             size_t length, int flags,
1880015             struct sockaddr *addrfrom, socklen_t * addrlen)
1880016 {
1880017   fd_t *sfd;
1880018   int i;        // IP table index.
1880019   size_t size_read;
1880020   struct udphdr *udp;
1880021   void *data;
1880022   struct sockaddr_in *addrfrom_in = (void *) addrfrom;
1880023   //
1880024   // Get file descriptor and verify that it is a
1880025   // socket.
1880026   //
1880027   sfd = fd_reference (pid, &sfdn);
1880028   if (sfd == NULL || sfd->file == NULL)
1880029     {
1880030       errset (EBADF);   // Bad file descriptor.
1880031       return ((ssize_t) - 1);
1880032     }
1880033   if (sfd->file->sock == NULL)
1880034     {
1880035       errset (ENOTSOCK);        // Not a socket.
1880036       return ((ssize_t) - 1);
1880037     }
1880038   //
1880039   // Verify to have a valid buffer pointer.
1880040   //
1880041   if (buffer == NULL)
1880042     {
1880043       errset (EINVAL);
```



```
1880044       return ((ssize_t) - 1);
1880045     }
1880046   //
1880047   //
1880048   //
1880049   if (sfd->file->sock->family == AF_INET)
1880050     {
1880051       //
1880052       // INET
1880053       //
1880054       // Should do some check here...
1880055       //
1880056       if (sfd->file->sock->type == SOCK_RAW)
1880057         {
1880058           //
1880059           // RAW
1880060           //
1880061           if (sfd->file->sock->protocol == IPPROTO_ICMP)
1880062             {
1880063               //
1880064               // ICMP
1880065               //
1880066               //
1880067               // Scan the ip_table[] to find an ICMP
1880068               // packet
1880069               // that was not already seen by the
1880070               // socket.
1880071               //
1880072               for (i = 0; i < IP_MAX_PACKETS; i++)
1880073                 {
1880074                   //
1880075                   // Check the protocol.
1880076                   //
1880077                   if (ip_table[i].packet.header.protocol !=
1880078                       IPPROTO_ICMP)
1880079                     {
1880080                       //
1880081                       // It is not ICMP.
1880082                       //
1880083                       continue;
1880084                     }
1880085                   //
1880086                   // Is the packet new for the socket?
1880087                   //
1880088                   // Please notice that the kernel
1880089                   // might be interrupted
1880090                   // also between clock tics; so,
1880091                   // during a single clock
1880092                   // time, a new packet might be
1880093                   // reached.
1880094                   //
1880095                   if (ip_table[i].clock
1880096                       < sfd->file->sock->read.clock[i])
1880097                     {
1880098                       //
1880099                       // Already seen or packet too
1880100                       // old.
1880101                       //
1880102                       continue;
1880103                     }
1880104                   //
1880105                   // Verify the IP addresses.
1880106                   //
1880107                   if (ip_table[i].packet.header.daddr
1880108                       != htonl (sfd->file->sock->laddr)
1880109                       && sfd->file->sock->laddr != 0)
1880110                     {
1880111                       //
1880112                       // The local address does not
1880113                       // match, and it is
1880114                       // not zero.
1880115                       //
1880116                       continue;
1880117                     }
1880118                   //
1880119                   if (ip_table[i].packet.header.saddr
1880120                       != htonl (sfd->file->sock->raddr))
1880121                     {
1880122                       //
1880123                       // The remote address does not
1880124                       // match, but
1880125                       // if it is zero, we accept all.
1880126                       //
1880127                       if (sfd->file->sock->raddr == 0)
1880128                         {
1880129                           //
1880130                           // Can accept the packet.
```

```
                        //
                        ;
                      }
                    else
                      {
                        continue;
                      }
                  }
                //
                // Packet accepted.
                //
                // This ICMP RAW packet is new for
                // the
                // socket: save the clock time, so
                // that the
                // same packet is not read again.
                //
                sfd->file->sock->read.clock[i]
                  = ip_table[i].clock;
                //
                // Copy the packet.
                //
                size_read
                  =
                  min (ntohs
                       (ip_table[i].packet.header.
                        tot_len), length);
                //
                memcpy (buffer,
                        ip_table[i].packet.octet,
                        size_read);
                //
                // Get the source address and
                // return.
                //
                if (addrfrom != NULL && addrlen != NULL)
                  {
                    if (*addrlen >=
                        sizeof (struct sockaddr_in))
                      {
                        addrfrom_in->sin_family = AF_INET;
                        addrfrom_in->sin_port = 0;
                        addrfrom_in->sin_addr.s_addr
                          =
                          ip_table[i].packet.header.saddr;
                      }
                    *addrlen =
                      sizeof (struct sockaddr_in);
                  }
                return ((ssize_t) size_read);
              }
          }
        else
          {
            //
            // Unsupported protocol.
            //
            errset (EPROTONOSUPPORT);
            return ((ssize_t) - 1);
          }
      }
    else if (sfd->file->sock->type == SOCK_DGRAM)
      {
        //
        // DGRAM
        //
        if (sfd->file->sock->protocol == IPPROTO_UDP)
          {
            //
            // UDP
            //
            // Scan the ip_table[] to find an UDP
            // packet
            // that was not already seen by the
            // socket.
            //
            for (i = 0; i < IP_MAX_PACKETS; i++)
              {
                //
                // Check the protocol.
                //
                if (ip_table[i].packet.header.protocol !=
                    IPPROTO_UDP)
                  {
                    //
                    // It is not UDP.
                    //
                    continue;
                  }
                //
                // Is the packet new for the socket?
                //
                // Please notice that the kernel
                // might be interrupted
                // also between clock tics; so,
                // during a single clock
                // time, a new packet might be
                // reached.
                //
                if (ip_table[i].clock
                    < sfd->file->sock->read.clock[i])
                  {
                    //
                    // Already seen or packet too
                    // old.
                    //
                    continue;
                  }
                //
                // Verify the ports.
                //
                udp = (struct udphdr *)
                  &ip_table[i].packet.octet
                  [sizeof (struct iphdr)];
                //
                if (udp->dest == 0)
                  {
                    //
                    // Cannot accept packets for the
                    // port zero!
                    //
                    continue;
                  }
                //
                if (udp->dest !=
                    htons (sfd->file->sock->lport))
                  {
                    //
                    // The local port does not
                    // match!
                    //
                    continue;
                  }
                //
                if (udp->source !=
                    htons (sfd->file->sock->rport)
                    && sfd->file->sock->rport != 0)
                  {
                    //
                    // The remote port does not
                    // match, and is not
                    // zero.
                    //
                    continue;
                  }
                //
                // Verify the IP addresses.
                //
                if (ip_table[i].packet.header.daddr
                    != htonl (sfd->file->sock->laddr)
                    && sfd->file->sock->laddr != 0)
                  {
                    //
                    // The local address does not
                    // match, and is
                    // not zero.
                    //
                    continue;
                  }
                //
                if (ip_table[i].packet.header.saddr
                    != htonl (sfd->file->sock->raddr)
                    && sfd->file->sock->raddr != 0)
                  {
                    //
                    // The remote address does not
                    // match, and is
                    // not zero.
                    //
                    continue;
                  }
                //
                // The packet is accepted.
                //
```

```
                  // This UDP packet is new for the
                  // socket:
                  // save the clock time, so that the
                  // same packet is not read again.
                  //
                  sfd->file->sock->read.clock[i]
                    = ip_table[i].clock;
                  //
                  // Check the right minimal size to
                  // be read, comparing
                  // the size of the IP packet, the
                  // size of the UDP
                  // packet and the size requested.
                  //
                  size_read =
                    ntohs (ip_table[i].packet.header.
                           tot_len) -
                    (ip_table[i].packet.header.ihl * 4) -
                    (sizeof (struct udphdr));
                  size_read =
                    min (size_read,
                         (udp->len -
                          sizeof (struct udphdr)));
                  size_read = min (size_read, length);
                  //
                  // Copy the data inside the UDP
                  // packet.
                  //
                  data =
                    (((uint8_t *) udp) +
                     sizeof (struct udphdr));
                  //
                  memcpy (buffer, data, size_read);
                  //
                  // Get the source address and
                  // return.
                  //
                  if (addrfrom != NULL && addrlen != NULL)
                    {
                      if (*addrlen >=
                          sizeof (struct sockaddr_in))
                        {
                          addrfrom_in->sin_family = AF_INET;
                          addrfrom_in->sin_port =
                            udp->source;
                          addrfrom_in->sin_addr.s_addr =
                            ip_table[i].packet.header.saddr;
                        }
                      *addrlen =
                        sizeof (struct sockaddr_in);
                    }
                  return ((ssize_t) size_read);
                }
            }
        }
      else if (sfd->file->sock->type == SOCK_STREAM)
        {
          //
          // STREAM
          //
          if (sfd->file->sock->protocol == IPPROTO_TCP)
            {
              //
              // TCP
              //
              // See if the read side of the stream
              // was closed.
              //
              if (sfd->file->sock->tcp.recv_closed
                  || sfd->file->sock->tcp.conn == TCP_CLOSE)
                {
                  //
                  // If the 'recv_size' is zero, the
                  // stream
                  // is closed.
                  //
                  if (sfd->file->sock->tcp.recv_size
                      == 0
                      || sfd->file->sock->tcp.can_read == 0)
                    {
                      //
                      // End of file.
                      //
                      return ((ssize_t) 0);
                    }
                }
              //
```

```
              // At the moment, nothing was read.
              //
              size_read = 0;
              //
              // See if there is data to be read from
              // the stream.
              //
              if (sfd->file->sock->tcp.can_read)
                {
                  size_read =
                    min (sfd->file->sock->tcp.recv_size,
                         length);
                  memcpy (buffer,
                          sfd->file->sock->tcp.recv_index,
                          size_read);
                  //
                  sfd->file->sock->tcp.recv_size -=
                    size_read;
                  sfd->file->sock->tcp.recv_index +=
                    size_read;
                  //
                  if (sfd->file->sock->tcp.recv_size == 0)
                    {
                      //
                      // Nothing to be read at the
                      // moment.
                      //
                      sfd->file->sock->tcp.can_read = 0;
                      sfd->file->sock->tcp.can_recv = 1;
                    }
                  //
                  // Get the source address and
                  // return.
                  //
                  if (addrfrom != NULL && addrlen != NULL)
                    {
                      if (*addrlen >=
                          sizeof (struct sockaddr_in))
                        {
                          addrfrom_in->sin_family = AF_INET;
                          addrfrom_in->sin_port =
                            htons (sfd->file->sock->rport);
                          addrfrom_in->sin_addr.s_addr =
                            htons (sfd->file->sock->raddr);
                        }
                      *addrlen =
                        sizeof (struct sockaddr_in);
                    }
                }
              //
              // Check if something was read.
              //
              if (size_read > 0)
                {
                  //
                  // Return normally.
                  //
                  return ((ssize_t) size_read);
                }
              else
                {
                  //
                  // Nothing to be read at the moment.
                  //
                  if (sfd->fl_flags & O_NONBLOCK)
                    {
                      //
                      // Try again.
                      //
                      errset (EAGAIN);
                      return ((ssize_t) - 1);
                    }
                  else
                    {
                      //
                      // Go to sleep and return a
                      // temporary error.
                      //
                      proc_table[pid].status =
                        PROC_SLEEPING;
                      proc_table[pid].ret = 0;
                      proc_table[pid].wakeup_events
                        = WAKEUP_EVENT_SOCK_READ;
                      proc_table[pid].wakeup_sock =
                        sfd->file->sock;
                      if (DEBUG)
                        {
```

```
                            k_printf
                              ("[%s:%i] PID %i goes to "
                               "sleep waiting to "
                               "receive for a socket.\n",
                               __FILE__, __LINE__, pid);
                          }
                        //
                        // Nothing was received.
                        //
                        errset (EAGAIN);
                        return ((ssize_t) - 1);
                      }
                  }
              }
            else
              {
                //
                // Unsupported protocol.
                //
                errset (EPROTONOSUPPORT);
                return ((ssize_t) - 1);
              }
          }
        else
          {
            //
            // Unsupported type.
            //
            errset (EPROTONOSUPPORT);
            return ((ssize_t) - 1);
          }
      }
    else
      {
        //
        // Unsupported family.
        //
        errset (EAFNOSUPPORT);
        return ((ssize_t) - 1);
      }
    //
    // If we are here, there are no more packets to read
    // at the moment.
    //
    if (sfd->fl_flags & O_NONBLOCK)
      {
        //
        // Try again.
        //
        errset (EAGAIN);
        return (-1);
      }
    else
      {
        //
        // The process should go to sleep.
        //
        proc_table[pid].status = PROC_SLEEPING;
        proc_table[pid].ret = 0;
        proc_table[pid].wakeup_events =
          WAKEUP_EVENT_SOCK_READ;
        proc_table[pid].wakeup_sock = sfd->file->sock;
        if (DEBUG)
          {
            k_printf ("[%s:%i] PID %i goes to sleep "
                      "waiting to receive "
                      "for a socket.\n",
                      __FILE__, __LINE__, pid);
          }
        //
        // Try again.
        //
        errset (EAGAIN);
        return ((ssize_t) - 1);
      }
}
```

94.8.31 kernel/lib_s/s_routeadd.c

«

Si veda la sezione 87.42.

```
#include <arpa/inet.h>
#include <sys/os32.h>
#include <kernel/net/route.h>
#include <kernel/lib_k.h>
#include <errno.h>
#include <netinet/in.h>
```



```
#include <kernel/proc.h>
//-----------------------------------------------------------
// This syscall is present only inside os32.
//-----------------------------------------------------------
int
s_routeadd (pid_t pid, in_addr_t dest, int m,
            in_addr_t router, int device)
{
  int r;
  h_addr_t netmask;
  h_addr_t network;
  //
  // Must be a privileged process.
  //
  if (proc_table[pid].euid != 0)
    {
      errset (EPERM);
      return (-1);
    }
  //
  //
  //
  if (m > 32 || m < 0)
    {
      errset (EINVAL);
      return (-1);
    }
  //
  // Calculate the netmask.
  //
  netmask = ip_mask (m);
  //
  // Fix the destination address, with the mask.
  //
  network = ntohl (dest) & netmask;
  //
  // Check if there is already. If there is: update
  // it.
  //
  for (r = 0; r < ROUTE_MAX_ROUTES; r++)
    {
      if (network == route_table[r].network
          && m == route_table[r].m)
        {
          //
          // Update.
          //
          route_table[r].router = ntohl (router);
          route_table[r].netmask = netmask;
          route_table[r].interface = device;
          return (0);
        }
    }
  //
  // The item is new. Find an empty place.
  //
  for (r = 0; r < ROUTE_MAX_ROUTES; r++)
    {
      if (route_table[r].network == 0xFFFFFFFF)
        {
          //
          // Empty.
          //
          route_table[r].network = network;
          route_table[r].netmask = netmask;
          route_table[r].m = m;
          route_table[r].router = ntohl (router);
          route_table[r].interface = device;
          //
          route_sort ();
          //
          return (0);
        }
    }
  //
  // No free space found.
  //
  errset (ENOMEM);
  return (-1);
}
```

94.8.32 kernel/lib_s/s_routedel.c

Si veda la sezione 87.43. «

```
#include <arpa/inet.h>
#include <sys/os32.h>
```

```
1900003 #include <kernel/net/route.h>
1900004 #include <kernel/lib_k.h>
1900005 #include <errno.h>
1900006 #include <netinet/in.h>
1900007 #include <kernel/proc.h>
1900008 //---------------------------------------------------------
1900009 // This syscall is present only inside os32.
1900010 //---------------------------------------------------------
1900011 int
1900012 s_routedel (pid_t pid, in_addr_t dest, int m)
1900013 {
1900014   int r;
1900015   h_addr_t network;
1900016   //
1900017   // Must be a privileged process.
1900018   //
1900019   if (proc_table[pid].euid != 0)
1900020     {
1900021       errset (EPERM);
1900022       return (-1);
1900023     }
1900024   //
1900025   //
1900026   //
1900027   if (m > 32 || m < 0)
1900028     {
1900029       errset (EINVAL);
1900030       return (-1);
1900031     }
1900032   //
1900033   // Calculate the destination network with the mask.
1900034   //
1900035   network = ntohl (dest) & ip_mask (m);
1900036   //
1900037   // Check if there is already. If there is: remove
1900038   // it.
1900039   //
1900040   for (r = 0; r < ROUTE_MAX_ROUTES; r++)
1900041     {
1900042       if (network == route_table[r].network
1900043           && m == route_table[r].m)
1900044         {
1900045           //
1900046           // Remove.
1900047           //
1900048           memset (&route_table[m], 0xFF,
1900049                   sizeof (route_table[m]));
1900050           return (0);
1900051         }
1900052     }
1900053   //
1900054   // Not found.
1900055   //
1900056   errset (EINVAL);
1900057   return (-1);
1900058 }
```

94.8.33 kernel/lib_s/s_sbrk.c

«

Si veda la sezione 87.5.

```
1910001 #include <errno.h>
1910002 #include <kernel/proc.h>
1910003 #include <kernel/lib_k.h>
1910004 #include <kernel/lib_s.h>
1910005 //---------------------------------------------------------
1910006 void *
1910007 s_sbrk (pid_t pid, intptr_t increment)
1910008 {
1910009   size_t previous_size;
1910010   size_t new_size;
1910011   int status;
1910012   //
1910013   // Get current data segment full size.
1910014   //
1910015   if (proc_table[pid].domain_data == 0)
1910016     {
1910017       previous_size = (proc_table[pid].domain_text
1910018                        + proc_table[pid].extra_data);
1910019     }
1910020   else
1910021     {
1910022       previous_size = (proc_table[pid].domain_data
1910023                        + proc_table[pid].extra_data);
1910024     }
1910025   //
1910026   // Check increment.
1910027   //
1910028   if ((increment + proc_table[pid].extra_data) < 0)
1910029     {
1910030       //
1910031       // Cannot reduce too much. Just correct it.
1910032       //
1910033       increment = -proc_table[pid].extra_data;
1910034     }
1910035   //
1910036   // Calculate the new size.
1910037   //
1910038   new_size = previous_size + increment;
1910039   //
1910040   // Call 's_brk()' to do the work. The new size value
1910041   // is the
1910042   // same of the new requested pointer address.
1910043   //
1910044   status = s_brk (pid, (void *) new_size);
1910045   //
1910046   if (status < 0)
1910047     {
1910048       errset (errno);
1910049       return ((void *) -1);
1910050     }
1910051   //
1910052   // Ok: return previous final address.
1910053   //
1910054   return ((void *) previous_size);
1910055 }
```


94.8.34 kernel/lib_s/s_send.c

«

Si veda la sezione 87.45.

```
1920001 #include <kernel/proc.h>
1920002 #include <kernel/net.h>
1920003 #include <kernel/net/route.h>
1920004 #include <kernel/net/udp.h>
1920005 #include <errno.h>
1920006 #include <arpa/inet.h>
1920007 #include <fcntl.h>
1920008 #include <sys/os32.h>
1920009 //---------------------------------------------------------
1920010 #define DEBUG 0
1920011 //---------------------------------------------------------
1920012 ssize_t
1920013 s_send (pid_t pid, int sfdn, const void *buffer,
1920014         size_t size, int flags)
1920015 {
1920016   fd_t *sfd;
1920017   int status;
1920018   //
1920019   // Get file descriptor and verify that it is a
1920020   // socket.
1920021   //
1920022   sfd = fd_reference (pid, &sfdn);
1920023   if (sfd == NULL || sfd->file == NULL)
1920024     {
1920025       errset (EBADF);   // Bad file descriptor.
1920026       return ((ssize_t) - 1);
1920027     }
1920028   if (sfd->file->sock == NULL)
1920029     {
1920030       errset (ENOTSOCK);        // Not a socket.
1920031       return ((ssize_t) - 1);
1920032     }
1920033   if (sfd->file->sock->unreach_port)
1920034     {
1920035       errset (ECONNREFUSED);    // Connection refused.
1920036       return ((ssize_t) - 1);
1920037     }
1920038   if (sfd->file->sock->unreach_prot)
1920039     {
1920040       errset (ENOPROTOOPT);     // Protocol not
1920041       // available.
1920042       return ((ssize_t) - 1);
1920043     }
1920044   if (sfd->file->sock->unreach_host)
1920045     {
1920046       errset (EHOSTUNREACH);    // Host unreachable.
1920047       return ((ssize_t) - 1);
1920048     }
1920049   if (sfd->file->sock->unreach_net)
1920050     {
1920051       errset (ENETUNREACH);     // Net unreachable.
1920052       return ((ssize_t) - 1);
1920053     }
```

```
  //
  // Verify to have a valid buffer pointer.
  //
  if (buffer == NULL)
    {
      errset (EINVAL);
      return ((ssize_t) - 1);
    }
  //
  //
  //
  if (sfd->file->sock->family == AF_INET)
    {
      //
      // INET
      //
      // AF_INET requires at least the remote address.
      //
      if (sfd->file->sock->raddr == 0)
        {
          errset (EDESTADDRREQ);
          return ((ssize_t) - 1);
        }
      //
      if (sfd->file->sock->type == SOCK_RAW)
        {
          //
          // RAW
          //
          if (sfd->file->sock->protocol == IPPROTO_ICMP)
            {
              //
              // ICMP
              //
              status = ip_tx (sfd->file->sock->laddr,
                              sfd->file->sock->raddr,
                              sfd->file->sock->protocol,
                              buffer, size);
              if (status)
                {
                  errset (errno);
                  return ((ssize_t) - 1);
                }
              else
                {
                  return ((ssize_t) size);
                }
            }
          else
            {
              //
              // Unsupported protocol.
              //
              errset (EPROTONOSUPPORT);
              return ((ssize_t) - 1);
            }
        }
      else if (sfd->file->sock->type == SOCK_DGRAM)
        {
          //
          // DGRAM
          //
          if (sfd->file->sock->protocol == IPPROTO_UDP)
            {
              //
              // UDP
              //
              status = udp_tx (sfd->file->sock->lport,
                               sfd->file->sock->rport,
                               sfd->file->sock->laddr,
                               sfd->file->sock->raddr,
                               buffer, size);
              if (status)
                {
                  errset (errno);
                  return ((ssize_t) - 1);
                }
              else
                {
                  return ((ssize_t) size);
                }
            }
          else
            {
              //
              // Unsupported protocol.
              //
              errset (EPROTONOSUPPORT);
              return ((ssize_t) - 1);
            }
        }
      else if (sfd->file->sock->type == SOCK_STREAM)
        {
          //
          // STREAM
          //
          if (sfd->file->sock->protocol == IPPROTO_TCP)
            {
              //
              // TCP
              //
              // See if the send side of the stream
              // was closed.
              //
              if (sfd->file->sock->tcp.send_closed
                  || sfd->file->sock->tcp.conn == TCP_CLOSE)
                {
                  //
                  // End of file.
                  //
                  if (DEBUG)
                    {
                      k_printf ("end of socket write\n");
                    }
                  s_kill ((pid_t) 0, pid, SIGPIPE);
                  errset (EPIPE);
                  return ((ssize_t) - 1);
                }
              //
              // Put data to the send buffer, if it is
              // possible.
              //
              if (sfd->file->sock->tcp.can_write)
                {
                  size =
                    min (size,
                         (TCP_MSS -
                          sizeof (struct tcphdr)));
                  memcpy (sfd->file->sock->tcp.send_data,
                          buffer, size);
                  sfd->file->sock->tcp.send_size = size;
                  sfd->file->sock->tcp.can_write = 0;
                  sfd->file->sock->tcp.can_send = 1;
                  //
                  sfd->file->sock->tcp.lsq[++sfd->
                                           file->sock->tcp.
                                           lsqi] =
                    sfd->file->sock->tcp.lsq_ack;
                  sfd->file->sock->tcp.send_flags =
                    TCP_FLAG_PSH | TCP_FLAG_ACK;
                  tcp_tx_sock (sfd->file->sock);
                  //
                  return ((ssize_t) size);
                }
              else
                {
                  //
                  // At the moment, nothing can be
                  // written.
                  //
                  if (sfd->fl_flags & O_NONBLOCK)
                    {
                      //
                      // Cannot block.
                      //
                      errset (EAGAIN);
                      return ((ssize_t) - 1);
                    }
                  else
                    {
                      //
                      // Go to sleep and return zero.
                      //
                      proc_table[pid].status =
                        PROC_SLEEPING;
                      proc_table[pid].ret = 0;
                      proc_table[pid].wakeup_events
                        = WAKEUP_EVENT_SOCK_WRITE;
                      proc_table[pid].wakeup_sock =
                        sfd->file->sock;
                      if (DEBUG)
                        {
                          k_printf
                            ("[%s] PID %i goes to "
```

```
                                "sleep waiting to write "
                                "to a socket.\n",
                                __FILE__, pid);
                          }
                        //
                        // Retry.
                        //
                        errset (EAGAIN);
                        return ((ssize_t) - 1);
                      }
                  }
              }
            else
              {
                //
                // Unsupported protocol.
                //
                errset (EPROTONOSUPPORT);
                return ((ssize_t) - 1);
              }
          }
        else
          {
            //
            // Unsupported type.
            //
            errset (EPROTONOSUPPORT);
            return ((ssize_t) - 1);
          }
      }
    else
      {
        //
        // Unsupported family.
        //
        errset (EAFNOSUPPORT);
        return ((ssize_t) - 1);
      }
}
```

94.8.35 kernel/lib_s/s_setegid.c

Si veda la sezione 87.48.

```
#include <kernel/proc.h>
#include <kernel/lib_s.h>
#include <errno.h>
//-----------------------------------------------------------
int
s_setegid (pid_t pid, gid_t egid)
{
  if ((proc_table[pid].euid == 0)
      || (proc_table[pid].egid == 0))
    {
      proc_table[pid].egid = egid;
      return (0);
    }
  else if (egid == proc_table[pid].egid)
    {
      return (0);
    }
  else if (egid == proc_table[pid].gid
           || egid == proc_table[pid].sgid)
    {
      proc_table[pid].egid = egid;
      return (0);
    }
  else
    {
      errset (EPERM);
      return (-1);
    }
}
```

94.8.36 kernel/lib_s/s_seteuid.c

Si veda la sezione 87.51.

```
#include <kernel/proc.h>
#include <kernel/lib_s.h>
#include <errno.h>
//-----------------------------------------------------------
int
s_seteuid (pid_t pid, uid_t euid)
{
  if (proc_table[pid].euid == 0)
    {
      proc_table[pid].euid = euid;
      return (0);
    }
  else if (euid == proc_table[pid].euid)
    {
      return (0);
    }
  else if (euid == proc_table[pid].uid
           || euid == proc_table[pid].suid)
    {
      proc_table[pid].euid = euid;
      return (0);
    }
  else
    {
      errset (EPERM);
      return (-1);
    }
}
```


94.8.37 kernel/lib_s/s_setgid.c

Si veda la sezione 87.48.

```
#include <kernel/proc.h>
#include <kernel/lib_s.h>
#include <errno.h>
//-----------------------------------------------------------
int
s_setgid (pid_t pid, gid_t gid)
{
  if ((proc_table[pid].euid == 0)
      || (proc_table[pid].egid == 0))
    {
      proc_table[pid].gid = gid;
      proc_table[pid].egid = gid;
      proc_table[pid].sgid = gid;
      return (0);
    }
  else if (gid == proc_table[pid].egid)
    {
      return (0);
    }
  else if (gid == proc_table[pid].gid
           || gid == proc_table[pid].sgid)
    {
      proc_table[pid].egid = gid;
      return (0);
    }
  else
    {
      errset (EPERM);
      return (-1);
    }
}
```

94.8.38 kernel/lib_s/s_setjmp.c

Si veda la sezione 87.49.

```
#include <kernel/lib_s.h>
#include <kernel/proc.h>
#include <errno.h>
#include <setjmp.h>
//-----------------------------------------------------------
extern uint32_t proc_stack_pointer;
//-----------------------------------------------------------
int
s_setjmp (pid_t pid, jmp_buf env)
{
  jmp_stack_t *sp;
  jmp_env_t *jmpenv;
  //
  // Find where is the process stack in memory, from
  // the kernel point
  // of view. Please notice that the current stack at
  // 'proc_stack_pointer' will be saved from the
  // scheduler inside
  // the process table, and current stack saved inside
  // the process
  // table is not up to date.
  //
  sp = ptr (pid, (void *) proc_stack_pointer);
  //
  // Translate the pointer 'env', to the kernel point
  // of view.
  //
```

```
1960028   jmpenv = ptr (pid, env);
1960029   //
1960030   // Save the process stack.
1960031   //
1960032   jmpenv->eax0 = sp->eax0;
1960033   jmpenv->ecx0 = sp->ecx0;
1960034   jmpenv->edx0 = sp->edx0;
1960035   jmpenv->ebx0 = sp->ebx0;
1960036   jmpenv->ebp0 = sp->ebp0;
1960037   jmpenv->esi0 = sp->esi0;
1960038   jmpenv->edi0 = sp->edi0;
1960039   jmpenv->ds0 = sp->ds0;
1960040   jmpenv->es0 = sp->es0;
1960041   jmpenv->fs0 = sp->fs0;
1960042   jmpenv->gs0 = sp->gs0;
1960043   jmpenv->eflags0 = sp->eflags0;
1960044   jmpenv->cs0 = sp->cs0;
1960045   jmpenv->eip0 = sp->eip0;
1960046   //
1960047   jmpenv->eip1 = sp->eip1;
1960048   jmpenv->syscallnr = sp->syscallnr;
1960049   jmpenv->msg_pointer = sp->msg_pointer;
1960050   jmpenv->msg_size = sp->msg_size;
1960051   jmpenv->env = sp->env;
1960052   jmpenv->ret = sp->ret;
1960053   jmpenv->ebp1 = sp->ebp1;
1960054   jmpenv->eip2 = sp->eip2;
1960055   //
1960056   // Save also the stack pointer!
1960057   //
1960058   jmpenv->esp0 = proc_stack_pointer;
1960059   //
1960060   return 0;
1960061 }
```

94.8.39 kernel/lib_s/s_setuid.c

«

Si veda la sezione 87.51.

```
1970001 #include <kernel/proc.h>
1970002 #include <kernel/lib_s.h>
1970003 #include <errno.h>
1970004 //-------------------------------------------------------
1970005 int
1970006 s_setuid (pid_t pid, uid_t uid)
1970007 {
1970008   if (proc_table[pid].euid == 0)
1970009     {
1970010       proc_table[pid].uid = uid;
1970011       proc_table[pid].euid = uid;
1970012       proc_table[pid].suid = uid;
1970013       return (0);
1970014     }
1970015   else if (uid == proc_table[pid].euid)
1970016     {
1970017       return (0);
1970018     }
1970019   else if (uid == proc_table[pid].uid
1970020            || uid == proc_table[pid].suid)
1970021     {
1970022       proc_table[pid].euid = uid;
1970023       return (0);
1970024     }
1970025   else
1970026     {
1970027       errset (EPERM);
1970028       return (-1);
1970029     }
1970030 }
```

94.8.40 kernel/lib_s/s_signal.c

«

Si veda la sezione 87.52.

```
1980001 #include <kernel/lib_s.h>
1980002 #include <kernel/proc.h>
1980003 #include <errno.h>
1980004 //-------------------------------------------------------
1980005 sighandler_t
1980006 s_signal (pid_t pid, int sig, sighandler_t handler,
1980007           uintptr_t wrapper)
1980008 {
1980009   unsigned long int flag = 1L << (sig - 1);
1980010   sighandler_t previous;
1980011   //
1980012   if (sig <= 0)
1980013     {
1980014       errset (EINVAL);
1980015       return (SIG_ERR);
1980016     }
1980017   //
1980018   if (proc_table[pid].sig_ignore & flag)
1980019     {
1980020       previous = SIG_IGN;
1980021     }
1980022   else if (proc_table[pid].sig_handler[sig] !=
1980023            (uintptr_t) NULL)
1980024     {
1980025       previous =
1980026         (sighandler_t) proc_table[pid].sig_handler[sig];
1980027     }
1980028   else
1980029     {
1980030       previous = SIG_DFL;
1980031     }
1980032   //
1980033   if (handler == SIG_DFL)
1980034     {
1980035       //
1980036       // Enable signal.
1980037       //
1980038       proc_table[pid].sig_ignore &= ~flag;
1980039       //
1980040       return (previous);
1980041     }
1980042   else if (handler == SIG_IGN)
1980043     {
1980044       //
1980045       // Disable signal.
1980046       //
1980047       proc_table[pid].sig_ignore |= flag;
1980048       //
1980049       return (previous);
1980050     }
1980051   else
1980052     {
1980053       //
1980054       // Enable signal, store the handler address and
1980055       // the
1980056       // handler-wrapper.
1980057       //
1980058       proc_table[pid].sig_ignore &= ~flag;
1980059       proc_table[pid].sig_handler[sig] =
1980060         (uintptr_t) handler;
1980061       proc_table[pid].sig_handler_wrapper = wrapper;
1980062       //
1980063       return (previous);
1980064     }
1980065 }
```


94.8.41 kernel/lib_s/s_socket.c

«

Si veda la sezione 87.54.

```
1990001 #include <kernel/proc.h>
1990002 #include <kernel/lib_s.h>
1990003 #include <kernel/lib_k.h>
1990004 #include <errno.h>
1990005 #include <fcntl.h>
1990006 #include <sys/socket.h>
1990007 #include <arpa/inet.h>
1990008 //-------------------------------------------------------
1990009 int
1990010 s_socket (pid_t pid, int family, int type, int protocol)
1990011 {
1990012   fd_t *fd;
1990013   int sfdn;
1990014   file_t *file;
1990015   sock_t *sock;
1990016   //
1990017   // Check supported family type.
1990018   //
1990019   if (family != AF_INET)
1990020     {
1990021       errset (EAFNOSUPPORT);
1990022       return (-1);
1990023     }
1990024   //
1990025   // Check supported communication type.
1990026   //
1990027   if (type == SOCK_RAW || type == SOCK_DGRAM
1990028       || type == SOCK_STREAM)
1990029     {
1990030       //
```

```
      // Ok.
      //
      ;
    }
  else
    {
      errset (EPROTONOSUPPORT);
      return (-1);
    }
  //
  // Check supported protocol type.
  //
  if (protocol == IPPROTO_ICMP
      || protocol == IPPROTO_UDP || protocol == IPPROTO_TCP)
    {
      //
      // Ok.
      //
      ;
    }
  else
    {
      errset (EPROTONOSUPPORT);
      return (-1);
    }
  //
  // If it is a raw socket, must be a privileged
  // process.
  //
  if (type == SOCK_RAW && proc_table[pid].euid != 0)
    {
      errset (EACCES);
      return (-1);
    }
  //
  // Find a free slot inside the sock_table[].
  //
  sock = sock_reference (-1);
  if (sock == NULL)
    {
      errset (ENFILE);
      return (-1);
    }
  //
  // Find a free slot inside the file table.
  //
  file = file_reference (-1);
  if (file == NULL)
    {
      errset (ENFILE);  // Too many files open in
      // system.
      return (-1);
    }
  //
  // Find a free slot inside the file descriptor
  // table.
  // Variable 'sfdn' will be modified by the call to
  // 'fd_reference()'.
  //
  sfdn = -1;
  fd = fd_reference (pid, &sfdn);
  if (fd == NULL)
    {
      errset (EMFILE);  // Too many open files.
      return (-1);
    }
  //
  // socket, system file and file descriptor ready:
  // reset and put data
  // inside them. Please notice that the
  // tcp.listen_queue[] array is
  // reset with all 0xFF, because zero is a valid file
  // descriptor
  // number.
  //
  memset (sock, 0, sizeof (sock_t));
  sock->active = 1;
  sock->family = family;
  sock->type = type;
  sock->protocol = protocol;
  sock->lport = 0;
  sock->laddr = 0;
  sock->rport = 0;
  sock->raddr = 0;
  memset (sock->read.clock, 0x00,
          sizeof (sock->read.clock));
  memset (sock->tcp.listen_queue, 0xFF,
          sizeof (sock->tcp.listen_queue));
  //
  file->references = 1;
  file->oflags = O_RDWR;
  file->inode = NULL;
  file->sock = sock;
  file->offset = 0;
  //
  fd->fl_flags = (O_RDWR | O_APPEND);
  fd->fd_flags = 0;
  fd->file = file;
  //
  // Return the file descriptor.
  //
  return (sfdn);
}
```

94.8.42 kernel/lib_s/s_stat.c

«

Si veda la sezione 87.55.

```
#include <kernel/fs.h>
#include <errno.h>
#include <kernel/proc.h>
#include <kernel/lib_s.h>
//-----------------------------------------------------------
int
s_stat (pid_t pid, const char *path, struct stat *buffer)
{
  proc_t *ps;
  inode_t *inode;
  //
  // Check path.
  //
  if (path == NULL || strlen (path) == 0)
    {
      errset (EINVAL);
      return (-1);
    }
  //
  // Get process.
  //
  ps = proc_reference (pid);
  //
  // Try to load the file inode.
  //
  inode = path_inode (pid, path);
  if (inode == NULL)
    {
      //
      // Cannot access the file: it does not exists or
      // permissions are
      // not sufficient. Variable 'errno' is set by
      // function
      // 'path_inode()'.
      //
      errset (errno);
      return (-1);
    }
  //
  // Inode loaded: update the buffer.
  //
  buffer->st_dev = inode->sb->device;
  buffer->st_ino = inode->ino;
  buffer->st_mode = inode->mode;
  buffer->st_nlink = inode->links;
  buffer->st_uid = inode->uid;
  buffer->st_gid = inode->gid;
  if (S_ISBLK (buffer->st_mode)
      || S_ISCHR (buffer->st_mode))
    {
      buffer->st_rdev = inode->direct[0];
    }
  else
    {
      buffer->st_rdev = 0;
    }
  buffer->st_size = inode->size;
  buffer->st_atime = inode->time;       // All times
  // are the
  // same for
  buffer->st_mtime = inode->time;       // Minix 1
  // file
  // system.
  buffer->st_ctime = inode->time;       //
  buffer->st_blksize = inode->sb->blksize;
  buffer->st_blocks = inode->blkcnt;
```

```
2000067   //
2000068   // If the inode is a device special file, the
2000069   // 'st_rdev' value is
2000070   // taken from the first direct zone (as of Minix 1
2000071   // organization).
2000072   //
2000073   if (S_ISBLK (inode->mode) || S_ISCHR (inode->mode))
2000074     {
2000075       buffer->st_rdev = inode->direct[0];
2000076     }
2000077   else
2000078     {
2000079       buffer->st_rdev = 0;
2000080     }
2000081   //
2000082   // Release the inode and return.
2000083   //
2000084   inode_put (inode);
2000085   //
2000086   // Return.
2000087   //
2000088   return (0);
2000089 }
```

94.8.43 kernel/lib_s/s_stime.c

«

Si veda la sezione 87.59.

```
2010001 #include <kernel/lib_s.h>
2010002 #include <kernel/proc.h>
2010003 #include <stddef.h>
2010004 #include <errno.h>
2010005 //-----------------------------------------------------------
2010006 extern clock_t _clock_time;     // uint64_t
2010007 //-----------------------------------------------------------
2010008 int
2010009 s_stime (pid_t pid, time_t * timer)
2010010 {
2010011   if (proc_table[pid].euid != 0)
2010012     {
2010013       errset (EPERM);
2010014       return (-1);
2010015     }
2010016   //
2010017   _clock_time = *timer * CLOCKS_PER_SEC;
2010018   return (0);
2010019 }
```

94.8.44 kernel/lib_s/s_tcgetattr.c

«

Si veda la sezione 87.58.

```
2020001 #include <kernel/fs.h>
2020002 #include <errno.h>
2020003 #include <kernel/proc.h>
2020004 #include <kernel/lib_s.h>
2020005 #include <termios.h>
2020006 #include <sys/types.h>
2020007 //-----------------------------------------------------------
2020008 int
2020009 s_tcgetattr (pid_t pid, int fdn, struct termios *termios_p)
2020010 {
2020011   file_t *file;
2020012   inode_t *inode;
2020013   dev_t device;
2020014   int t;        // 'tty_table[]' subscript.
2020015   int c;        // 'c_cc[]' subscript.
2020016   //
2020017   file = proc_table[pid].fd[fdn].file;
2020018   //
2020019   if (file == NULL)
2020020     {
2020021       errset (EBADF);
2020022       return (-1);
2020023     }
2020024   //
2020025   inode = proc_table[pid].fd[fdn].file->inode;
2020026   //
2020027   if (inode == NULL)
2020028     {
2020029       errset (EBADF);
2020030       return (-1);
2020031     }
2020032   //
2020033   if (!S_ISCHR (inode->mode))
2020034     {
2020035       errset (ENOTTY);
2020036       return (-1);
2020037     }
2020038   //
2020039   device = inode->direct[0];
2020040   //
2020041   if (major (device) != DEV_CONSOLE_MAJOR)
2020042     {
2020043       errset (ENOTTY);
2020044       return (-1);
2020045     }
2020046   //
2020047   t = minor (device);
2020048   if (t >= TTYS_TOTAL)
2020049     {
2020050       errset (ENOTTY);
2020051       return (-1);
2020052     }
2020053   //
2020054   // Ok: copy data.
2020055   //
2020056   termios_p->c_iflag = tty_table[t].attr.c_iflag;
2020057   termios_p->c_oflag = tty_table[t].attr.c_oflag;
2020058   termios_p->c_cflag = tty_table[t].attr.c_cflag;
2020059   termios_p->c_lflag = tty_table[t].attr.c_lflag;
2020060   for (c = 0; c < NCCS; c++)
2020061     {
2020062       termios_p->c_cc[c] = tty_table[t].attr.c_cc[c];
2020063     }
2020064   //
2020065   // Ok.
2020066   //
2020067   return (0);
2020068 }
```


94.8.45 kernel/lib_s/s_tcsetattr.c

«

Si veda la sezione 87.58.

```
2030001 #include <kernel/fs.h>
2030002 #include <errno.h>
2030003 #include <kernel/proc.h>
2030004 #include <kernel/lib_s.h>
2030005 #include <termios.h>
2030006 #include <sys/types.h>
2030007 //-----------------------------------------------------------
2030008 // The following are masks of the implemented
2030009 // attributes.
2030010 //
2030011 #define MASK_C_IFLAG (BRKINT|ICRNL|IGNBRK|IGNCR)
2030012 #define MASK_C_OFLAG 0
2030013 #define MASK_C_CFLAG 0
2030014 #define MASK_C_LFLAG \
2030015     (ECHO|ECHOE|ECHOK|ECHONL|ICANON|ISIG)
2030016 //-----------------------------------------------------------
2030017 int
2030018 s_tcsetattr (pid_t pid, int fdn, int action,
2030019              struct termios *termios_p)
2030020 {
2030021   file_t *file;
2030022   inode_t *inode;
2030023   dev_t device;
2030024   int t;        // 'tty_table[]' subscript.
2030025   int c;        // 'c_cc[]' subscript.
2030026   //
2030027   file = proc_table[pid].fd[fdn].file;
2030028   //
2030029   if (file == NULL)
2030030     {
2030031       errset (EBADF);
2030032       return (-1);
2030033     }
2030034   //
2030035   inode = proc_table[pid].fd[fdn].file->inode;
2030036   //
2030037   if (inode == NULL)
2030038     {
2030039       errset (EBADF);
2030040       return (-1);
2030041     }
2030042   //
2030043   if (!S_ISCHR (inode->mode))
2030044     {
2030045       errset (ENOTTY);
2030046       return (-1);
2030047     }
2030048   //
2030049   device = inode->direct[0];
```

```
2030050   //
2030051   if (major (device) != DEV_CONSOLE_MAJOR)
2030052     {
2030053       errset (ENOTTY);
2030054       return (-1);
2030055     }
2030056   //
2030057   t = minor (device);
2030058   if (t >= TTYS_TOTAL)
2030059     {
2030060       errset (ENOTTY);
2030061       return (-1);
2030062     }
2030063   //
2030064   // The parameter 'actions' is silently ignored: only
2030065   // immediate update will take place.
2030066   //
2030067   // The function will not notice if at least a
2030068   // successful attribute change is done, so,
2030069   // after this point, the return value is
2030070   // always zero.
2030071   //
2030072   tty_table[t].attr.c_iflag =
2030073     (termios_p->c_iflag & MASK_C_IFLAG);
2030074   tty_table[t].attr.c_oflag =
2030075     (termios_p->c_oflag & MASK_C_OFLAG);
2030076   tty_table[t].attr.c_cflag =
2030077     (termios_p->c_cflag & MASK_C_CFLAG);
2030078   tty_table[t].attr.c_lflag =
2030079     (termios_p->c_lflag & MASK_C_LFLAG);
2030080   for (c = 0; c < NCCS; c++)
2030081     {
2030082       //
2030083       // Should be done some check here?
2030084       //
2030085       tty_table[t].attr.c_cc[c] = termios_p->c_cc[c];
2030086     }
2030087   //
2030088   // Ok.
2030089   //
2030090   return (0);
2030091 }
```

94.8.46 kernel/lib_s/s_time.c

«

Si veda la sezione 87.59.

```
2040001 #include <kernel/lib_k.h>
2040002 #include <kernel/lib_s.h>
2040003 #include <stddef.h>
2040004 //-------------------------------------------------------
2040005 extern clock_t _clock_time;     // uint64_t
2040006 //-------------------------------------------------------
2040007 time_t
2040008 s_time (pid_t pid, time_t * timer)
2040009 {
2040010   time_t time = _clock_time / CLOCKS_PER_SEC;
2040011   if (timer != NULL)
2040012     {
2040013       *timer = time;
2040014     }
2040015   return (time);
2040016 }
```

94.8.47 kernel/lib_s/s_umount.c

«

Si veda la sezione 87.36.

```
2050001 #include <kernel/fs.h>
2050002 #include <errno.h>
2050003 #include <kernel/proc.h>
2050004 #include <kernel/lib_s.h>
2050005 //-------------------------------------------------------
2050006 int
2050007 s_umount (pid_t pid, const char *path_mnt)
2050008 {
2050009   proc_t *ps;
2050010   dev_t device; // Device to mount.
2050011   inode_t *inode_mount_point;   // Original mount
2050012   // point.
2050013   inode_t *inode;       // Inode table.
2050014   int i;        // Inode table index.
2050015   //
2050016   // Get process.
2050017   //
2050018   ps = proc_reference (pid);
2050019   //
```



```
2050020   // Verify to be the super user.
2050021   //
2050022   if (ps->euid != 0)
2050023     {
2050024       errset (EPERM);   // Operation not permitted.
2050025       return (-1);
2050026     }
2050027   //
2050028   // Get the directory mount point.
2050029   //
2050030   inode_mount_point = path_inode (pid, path_mnt);
2050031   if (inode_mount_point == NULL)
2050032     {
2050033       errset (ENOENT);  // No such file or directory.
2050034       return (-1);
2050035     }
2050036   //
2050037   // Verify that the path is a directory.
2050038   //
2050039   if (!S_ISDIR (inode_mount_point->mode))
2050040     {
2050041       inode_put (inode_mount_point);
2050042       errset (ENOTDIR); // Not a directory.
2050043       return (-1);
2050044     }
2050045   //
2050046   // Verify that there is something attached.
2050047   //
2050048   device = inode_mount_point->sb_attached->device;
2050049   if (device == 0)
2050050     {
2050051       //
2050052       // There is nothing to unmount.
2050053       //
2050054       inode_put (inode_mount_point);
2050055       errset (E_NOT_MOUNTED);   // Not mounted.
2050056       return (-1);
2050057     }
2050058   //
2050059   // Are there exactly two internal references? Let's
2050060   // explain:
2050061   // the directory that act as mount point, should
2050062   // have one reference
2050063   // because it is mounting something and another
2050064   // because it was just
2050065   // opened again, a few lines above. If there are
2050066   // more references
2050067   // it is wrong; if there are less, it is also wrong
2050068   // at this point.
2050069   //
2050070   if (inode_mount_point->references != 2)
2050071     {
2050072       inode_put (inode_mount_point);
2050073       errset (EUNKNOWN);        // Unknown error.
2050074       return (-1);
2050075     }
2050076   //
2050077   // All data is available: find if there are open
2050078   // file inside
2050079   // the file system to unmount. But first load the
2050080   // inode table
2050081   // pointer.
2050082   //
2050083   inode = inode_reference ((dev_t) 0, (ino_t) 0);
2050084   if (inode == NULL)
2050085     {
2050086       //
2050087       // This error should not happen.
2050088       //
2050089       inode_put (inode_mount_point);
2050090       errset (EUNKNOWN);        // Unknown error.
2050091       return (-1);
2050092     }
2050093   //
2050094   // Scan the inode table.
2050095   //
2050096   for (i = 0; i < INODE_MAX_SLOTS; i++)
2050097     {
2050098       if ((inode[i].sb ==
2050099            inode_mount_point->sb_attached)
2050100           && (inode[i].references > 0))
2050101         {
2050102           //
2050103           // At least one file is open inside the
2050104           // super block to
2050105           // release: cannot unmount.
2050106           //
```

```
2050107           inode_put (inode_mount_point);
2050108           errset (EBUSY);       // Device or resource
2050109           // busy.
2050110           return (-1);
2050111         }
2050112     }
2050113   //
2050114   // Can unmount: save and remove the super block
2050115   // memory;
2050116   // clear the mount point reference and put inode.
2050117   //
2050118   inode_mount_point->sb_attached->changed = 1;
2050119   sb_save (inode_mount_point->sb_attached);
2050120   //
2050121   inode_mount_point->sb_attached->device = 0;
2050122   inode_mount_point->sb_attached->inode_mounted_on = NULL;
2050123   inode_mount_point->sb_attached->blksize = 0;
2050124   inode_mount_point->sb_attached->options = 0;
2050125   //
2050126   inode_mount_point->sb_attached = NULL;
2050127   inode_mount_point->references = 0;
2050128   inode_put (inode_mount_point);
2050129   //
2050130   inode_put (inode_mount_point);
2050131   //
2050132   return (0);
2050133 }
```

94.8.48 kernel/lib_s/s_unlink.c

«

Si veda la sezione 87.62.

```
2060001 #include <kernel/fs.h>
2060002 #include <errno.h>
2060003 #include <kernel/proc.h>
2060004 #include <libgen.h>
2060005 #include <kernel/lib_s.h>
2060006 #include <kernel/lib_k.h>
2060007 //-------------------------------------------------------
2060008 int
2060009 s_unlink (pid_t pid, const char *path)
2060010 {
2060011   proc_t *ps;
2060012   inode_t *inode_unlink;
2060013   inode_t *inode_directory;
2060014   char path_unlink[PATH_MAX];
2060015   char path_copy[PATH_MAX];
2060016   char *path_directory;
2060017   char *name_unlink;
2060018   dev_t device;
2060019   off_t start;
2060020   char buffer[SB_MAX_ZONE_SIZE];
2060021   directory_t *dir = (directory_t *) buffer;
2060022   int status;
2060023   ssize_t size_read;
2060024   ssize_t size_written;
2060025   int d;        // Directory buffer index.
2060026   //
2060027   // Get process.
2060028   //
2060029   ps = proc_reference (pid);
2060030   //
2060031   // Get full paths.
2060032   //
2060033   path_full (path, ps->path_cwd, path_unlink);
2060034   strncpy (path_copy, path_unlink, PATH_MAX);
2060035   path_directory = dirname (path_copy);
2060036   //
2060037   // Get the inode to be unlinked.
2060038   //
2060039   inode_unlink = path_inode (pid, path_unlink);
2060040   if (inode_unlink == NULL)
2060041     {
2060042       return (-1);
2060043     }
2060044   //
2060045   // If it is a directory, verify that it is empty.
2060046   //
2060047   if (S_ISDIR (inode_unlink->mode))
2060048     {
2060049       if (!inode_dir_empty (inode_unlink))
2060050         {
2060051           inode_put (inode_unlink);
2060052           errset (ENOTEMPTY);   // Directory not
2060053           // empty.
2060054           return (-1);
2060055         }
```



```
2060056     }
2060057   //
2060058   // Get the inode of the directory containing it.
2060059   //
2060060   inode_directory = path_inode (pid, path_directory);
2060061   if (inode_directory == NULL)
2060062     {
2060063       inode_put (inode_unlink);
2060064       return (-1);
2060065     }
2060066   //
2060067   // Check if something is mounted on the directory.
2060068   //
2060069   if (inode_directory->sb_attached != NULL)
2060070     {
2060071       //
2060072       // Must select the right directory.
2060073       //
2060074       device = inode_directory->sb_attached->device;
2060075       inode_put (inode_directory);
2060076       inode_directory = inode_get (device, 1);
2060077       if (inode_directory == NULL)
2060078         {
2060079           inode_put (inode_unlink);
2060080           return (-1);
2060081         }
2060082     }
2060083   //
2060084   // Check if write is allowed for the file system.
2060085   //
2060086   if (inode_directory->sb->options & MOUNT_RO)
2060087     {
2060088       errset (EROFS);   // Read-only file system.
2060089       return (-1);
2060090     }
2060091   //
2060092   // Verify access permissions for the directory. The
2060093   // number "3" means
2060094   // that the user must have access permission and
2060095   // write permission:
2060096   // "-wx" == 2+1 == 3.
2060097   //
2060098   status = inode_check (inode_directory, S_IFDIR, 3,
2060099                         ps->euid, ps->egid);
2060100   if (status != 0)
2060101     {
2060102       errset (EPERM);   // Operation not permitted.
2060103       inode_put (inode_unlink);
2060104       inode_put (inode_directory);
2060105       return (-1);
2060106     }
2060107   //
2060108   // Get the base name to be unlinked: this will alter
2060109   // the
2060110   // original path.
2060111   //
2060112   name_unlink = basename (path_unlink);
2060113   //
2060114   // Read the directory content and try to locate the
2060115   // item to unlink.
2060116   //
2060117   for (start = 0;
2060118        start < inode_directory->size;
2060119        start += inode_directory->sb->blksize)
2060120     {
2060121       size_read =
2060122         inode_file_read (inode_directory, start,
2060123                          buffer,
2060124                          inode_directory->sb->blksize,
2060125                          NULL);
2060126       if (size_read < sizeof (directory_t))
2060127         {
2060128           break;
2060129         }
2060130       //
2060131       // Scan the directory portion just read, for the
2060132       // item to unlink.
2060133       //
2060134       dir = (directory_t *) buffer;
2060135       //
2060136       for (d = 0; d < size_read;
2060137            d += (sizeof (directory_t)), dir++)
2060138         {
2060139           if ((dir->ino != 0)
2060140               &&
2060141               (strncmp
2060142                (dir->name, name_unlink, NAME_MAX) == 0))
```

```
2060143             {
2060144               //
2060145               // Found the corresponding item: unlink
2060146               // the inode.
2060147               //
2060148               dir->ino = 0;
2060149               //
2060150               // Update the directory inside the file
2060151               // system.
2060152               //
2060153               size_written =
2060154                 inode_file_write (inode_directory,
2060155                                   start, buffer, size_read);
2060156               if (size_written != size_read)
2060157                 {
2060158                   //
2060159                   // Write problem: just tell.
2060160                   //
2060161                   k_printf
2060162                     ("kernel alert: directory "
2060163                      "write error!\n");
2060164                 }
2060165               //
2060166               // Update directory inode and put inode.
2060167               // If the unlinked
2060168               // inode was a directory, the parent
2060169               // directory inode
2060170               // must reduce the file system link
2060171               // count.
2060172               //
2060173               if (S_ISDIR (inode_unlink->mode))
2060174                 {
2060175                   inode_directory->links--;
2060176                 }
2060177               inode_directory->time = s_time (pid, NULL);
2060178               inode_directory->changed = 1;
2060179               inode_put (inode_directory);
2060180               //
2060181               // Reduce link inside unlinked inode and
2060182               // put inode.
2060183               //
2060184               inode_unlink->links--;
2060185               inode_unlink->changed = 1;
2060186               inode_unlink->time = s_time (pid, NULL);
2060187               inode_put (inode_unlink);
2060188               //
2060189               // Just return, as the work is done.
2060190               //
2060191               return (0);
2060192             }
2060193         }
2060194     }
2060195   //
2060196   // At this point, it was not possible to unlink the
2060197   // file.
2060198   //
2060199   inode_put (inode_unlink);
2060200   inode_put (inode_directory);
2060201   errset (EUNKNOWN);    // Unknown error.
2060202   return (-1);
2060203 }
```

94.8.49 kernel/lib_s/s_wait.c

« Si veda la sezione 87.63.

```
2070001 #include <kernel/proc.h>
2070002 #include <kernel/lib_s.h>
2070003 #include <errno.h>
2070004 //-----------------------------------------------------------
2070005 pid_t
2070006 s_wait (pid_t pid, int *status)
2070007 {
2070008   pid_t parent = pid;
2070009   pid_t child;
2070010   int child_available = 0;
2070011   //
2070012   // Find a dead child process.
2070013   //
2070014   for (child = 1; child < PROCESS_MAX; child++)
2070015     {
2070016       if (proc_table[child].ppid == parent)
2070017         {
2070018           child_available = 1;  // Child found!
2070019           if (proc_table[child].status == PROC_ZOMBIE)
2070020             {
2070021               break;    // It is dead!
2070022             }
2070023         }
2070024     }
2070025   //
2070026   // If the index 'child' is a valid process number,
2070027   // a dead child was found.
2070028   //
2070029   if (child < PROCESS_MAX)
2070030     {
2070031       *status = proc_table[child].ret;
2070032       proc_available (child);
2070033       return (child);
2070034     }
2070035   else
2070036     {
2070037       if (child_available)
2070038         {
2070039           //
2070040           // There are child, but all alive.
2070041           //
2070042           // Go to sleep.
2070043           //
2070044           proc_table[parent].status = PROC_SLEEPING;
2070045           proc_table[parent].wakeup_events =
2070046             WAKEUP_EVENT_SIGNAL;
2070047           proc_table[parent].wakeup_signal = SIGCHLD;
2070048           return ((pid_t) 0);
2070049         }
2070050       else
2070051         {
2070052           //
2070053           // There are no child at all.
2070054           //
2070055           errset (ECHILD);
2070056           return ((pid_t) - 1);
2070057         }
2070058     }
2070059 }
```

94.8.50 kernel/lib_s/s_write.c

« Si veda la sezione 87.64.

```
2080001 #include <kernel/proc.h>
2080002 #include <kernel/lib_s.h>
2080003 #include <errno.h>
2080004 #include <fcntl.h>
2080005 //-----------------------------------------------------------
2080006 #define DEBUG 0
2080007 //-----------------------------------------------------------
2080008 ssize_t
2080009 s_write (pid_t pid, int fdn, const void *buffer,
2080010          size_t count)
2080011 {
2080012   proc_t *ps;
2080013   fd_t *fd;
2080014   ssize_t size_written;
2080015   int status;
2080016   //
2080017   // Get process.
2080018   //
2080019   ps = proc_reference (pid);
2080020   //
2080021   // Get file descriptor.
2080022   //
2080023   fd = fd_reference (pid, &fdn);
2080024   if (fd == NULL || fd->file == NULL
2080025       || (fd->file->inode == NULL
2080026           && fd->file->sock == NULL))
2080027     {
2080028       //
2080029       // The file descriptor pointer is not valid.
2080030       //
2080031       errset (EBADF);   // Bad file descriptor.
2080032       return ((ssize_t) - 1);
2080033     }
2080034   //
2080035   // Check if it is opened for write.
2080036   //
2080037   if (!(fd->file->oflags & O_WRONLY))
2080038     {
2080039       //
2080040       // The file is not opened for write.
2080041       //
2080042       errset (EINVAL);  // Invalid argument.
2080043       return ((ssize_t) - 1);
2080044     }
```

```
2080045   //
2080046   // Check if it is a directory inode: a directory can
2080047   // be
2080048   // read as a file descriptor, but cannot be written.
2080049   //
2080050   if (fd->file->inode != NULL
2080051       && (fd->file->inode->mode & S_IFDIR))
2080052     {
2080053       errset (EISDIR);  // Is a directory.
2080054       return ((ssize_t) - 1);
2080055     }
2080056   //
2080057   // It should be a valid type of file or socket to be
2080058   // written.
2080059   // Check if it is a file opened in append mode: if
2080060   // so, must move
2080061   // the write offset to the end.
2080062   //
2080063   if (fd->file->inode != NULL && (fd->fl_flags & O_APPEND))
2080064     {
2080065       fd->file->offset = fd->file->inode->size;
2080066     }
2080067   //
2080068   // Check the kind of socket/file to be written and
2080069   // write it.
2080070   //
2080071   if (fd->file->sock != NULL)
2080072     {
2080073       //
2080074       // Send it.
2080075       //
2080076       size_written = s_send (pid, fdn, buffer, count, 0);
2080077     }
2080078   else if (fd->file->inode->mode & S_IFBLK ||
2080079            fd->file->inode->mode & S_IFCHR)
2080080     {
2080081       //
2080082       // A device is to be written.
2080083       //
2080084       size_written =
2080085         dev_io (pid,
2080086                 (dev_t) fd->file->inode->direct[0],
2080087                 DEV_WRITE, (off_t) fd->file->offset,
2080088                 (void *) buffer, count, NULL);
2080089     }
2080090   else if (fd->file->inode->mode & S_IFREG)
2080091     {
2080092       //
2080093       // A regular file is to be written.
2080094       //
2080095       size_written = inode_file_write (fd->file->inode,
2080096                                        fd->file->offset,
2080097                                        buffer, count);
2080098     }
2080099   else if (fd->file->inode->mode & S_IFIFO)
2080100     {
2080101       //
2080102       // A pipe is to be written.
2080103       //
2080104       size_written =
2080105         inode_pipe_write (fd->file->inode, buffer, count);
2080106       if (size_written == 0)
2080107         {
2080108           if (fd->file->inode->pipe_ref_read == 0)
2080109             {
2080110               //
2080111               // No read will be done anymore. Tell to
2080112               // the process.
2080113               //
2080114               status = s_kill ((pid_t) 0, pid, SIGPIPE);
2080115               if (status < 0)
2080116                 {
2080117                   errset (EPIPE);
2080118                   return ((ssize_t) - 1);
2080119                 }
2080120               else
2080121                 {
2080122                   //
2080123                   // Wake up processes waiting to
2080124                   // read.
2080125                   //
2080126                   proc_wakeup_pipe_read (fd->file->inode);
2080127                 }
2080128             }
2080129           else
2080130             {
2080131               //
```



```
2080132               // At the moment, nothing can be
2080133               // written.
2080134               //
2080135               if (fd->fl_flags & O_NONBLOCK)
2080136                 {
2080137                   //
2080138                   // Cannot block.
2080139                   //
2080140                   errset (EAGAIN);
2080141                   return ((ssize_t) - 1);
2080142                 }
2080143               else
2080144                 {
2080145                   //
2080146                   // Go to sleep.
2080147                   //
2080148                   proc_table[pid].status = PROC_SLEEPING;
2080149                   proc_table[pid].ret = 0;
2080150                   proc_table[pid].wakeup_inode =
2080151                     fd->file->inode;
2080152                   proc_table[pid].wakeup_events =
2080153                     WAKEUP_EVENT_PIPE_WRITE;
2080154                   if (DEBUG)
2080155                     {
2080156                       k_printf
2080157                         ("[%s] PID %i goes to sleep "
2080158                          "waiting to write inside "
2080159                          "a pipe.\n", __FILE__, pid);
2080160                     }
2080161                 }
2080162             }
2080163         }
2080164       else
2080165         {
2080166           //
2080167           // Wake up processes waiting to read.
2080168           //
2080169           proc_wakeup_pipe_read (fd->file->inode);
2080170         }
2080171     }
2080172   else
2080173     {
2080174       //
2080175       // Unsupported file type.
2080176       //
2080177       errset (E_FILE_TYPE_UNSUPPORTED); // File type
2080178       // unsupported.
2080179       return ((ssize_t) - 1);
2080180     }
2080181   //
2080182   // Update the file descriptor internal offset, but
2080183   // only if it
2080184   // is a file.
2080185   //
2080186   if (fd->file->sock == NULL && size_written > 0)
2080187     {
2080188       fd->file->offset += size_written;
2080189     }
2080190   //
2080191   // Just return the size written, even if it is an
2080192   // error.
2080193   //
2080194   return (size_written);
2080195 }
```

94.9 os32: «kernel/main.h»

Si veda la sezione 93.13.

```
2090001 #ifndef _KERNEL_MAIN_H
2090002 #define _KERNEL_MAIN_H       1
2090003 //*************************************************************
2090004 #include <kernel/multiboot.h>
2090005 #include <stdint.h>
2090006 #include <sys/types.h>
2090007 //*************************************************************
2090008 void kmain (uint32_t magic, multiboot_t * mboot_data);
2090009 void menu (void);
2090010 pid_t run (char *path, char *argv[], char *envp[]);
2090011 //*************************************************************
2090012 #endif
```

94.9.1 kernel/main/build.h

Si veda la sezione 93.13.

```
2100001 #define BUILD_DATE "201108311936"
```

94.9.2 kernel/main/crt0.s

Si veda la sezione 84.2.2.

```
2110001 .extern kmain
2110002 .extern _k_stack_top
2110003 .extern _k_stack_bottom
2110004 .global kstartup
2110005 ########################################################
2110006 #
2110007 # The kernel must be compiled as ELF, so that the
2110008 # bootloader (GRUB or SYSLINUX) can recognize it.
2110009 #
2110010 #-------------------------------------------------------
2110011 .section .text
2110012 #-------------------------------------------------------
2110013 #
2110014 # At the beginning there is the code, but inside the
2110015 # code there is also the multiboot header.
2110016 # This is why we start with a jump.
2110017 #
2110018 kstartup:
2110019     jmp start
2110020 #
2110021 # Here is the multiboot header, that must be placed
2110022 # near the beginning of the image-file, but aligned at
2110023 # a multiple of four bytes.
2110024 #
2110025 .align 4
2110026 multiboot_header:
2110027     .int 0x1BADB002                     # magic
2110028     .int 0x00000007                     # flags
2110029     .int -(0x1BADB002 + 0x00000007)     # checksum
2110030 #
2110031 # Here starts really the code.
2110032 #
2110033 start:
2110034     #
2110035     # Set ESP at the stack bottom.
2110036     #
2110037     movl $_k_stack_bottom, %esp
2110038     #
2110039     # Reset flags inside EFLAGS, but must use the stack
2110040     # to make it.
2110041     #
2110042     pushl $0
2110043     popf
2110044     #
2110045     # Call function 'kmain()'. It is not the usual
2110046     # 'main()' because we need to pass some data, but
2110047     # it is not compatible with the
2110048     # standard 'main()' prototype.
2110049     #
2110050     # void kmain (uint32_t magic, multiboot_t *info);
2110051     #
2110052     pushl %ebx    # Pointer to the structure containing
2110053                   # data from the boot system.
2110054     pushl %eax    # Boot system signature.
2110055     #
2110056     call kmain    # Do the call.
2110057 #
2110058 # Halt procedure.
2110059 #
2110060 halt:
2110061     hlt         # If the function 'kmain()' return, this
2110062                 # instruction will halt the CPU.
2110063     jmp halt    # If the CPU will resume working, the
2110064                 # HLT instruction will be repeated.
2110065 #-------------------------------------------------------
2110066 .align 4
2110067 .section .data
2110068 #-------------------------------------------------------
2110069 .align 4
2110070 .section .bss
2110071 #-------------------------------------------------------
```


94.9.3 kernel/main/kmain.c

Si veda la sezione 93.13.

```
2120001 #include <kernel/main.h>
2120002 #include <kernel/main/build.h>
2120003 #include <kernel/lib_k.h>
2120004 #include <kernel/driver/tty.h>
2120005 #include <kernel/memory.h>
2120006 #include <stdlib.h>
2120007 #include <stdint.h>
2120008 #include <kernel/driver/screen.h>
2120009 #include <kernel/proc.h>
2120010 #include <kernel/lib_s.h>
2120011 #include <kernel/fs.h>
2120012 #include <unistd.h>
2120013 #include <stdint.h>
2120014 #include <kernel/driver/kbd.h>
2120015 #include <kernel/driver/ata.h>
2120016 #include <kernel/dev.h>
2120017 #include <kernel/blk.h>
2120018 #include <fcntl.h>
2120019 #include <string.h>
2120020 #include <limits.h>
2120021 #include <kernel/driver/pci.h>
2120022 #include <kernel/net/icmp.h>
2120023 #include <kernel/net/arp.h>
2120024 #include <kernel/net/route.h>
2120025 #include <kernel/net/tcp.h>
2120026 #include <kernel/driver/nic/ne2k.h>
2120027 #include <errno.h>
2120028 //-------------------------------------------------------
2120029 static char command[MAX_CANON];
2120030 //-------------------------------------------------------
2120031 void
2120032 kmain (uint32_t magic, multiboot_t * mboot_data)
2120033 {
2120034   int exit;
2120035   pid_t pid;
2120036   int counter;
2120037   char *exec_argv[2];
2120038   int status;
2120039   ssize_t count;
2120040   //
2120041   // 0xAC15FEFE == 172.21.254.254
2120042   //
2120043   h_addr_t ip_to_be_found = 0xAC15FEFE;
2120044   //
2120045   // Reset video and select the initial console.
2120046   //
2120047   tty_init ();
2120048   //
2120049   // Verify 'multiboot' data.
2120050   //
2120051   if (magic == 0x2BADB002)
2120052     {
2120053       //
2120054       // Save multiboot data.
2120055       //
2120056       mboot_save (mboot_data);
2120057       //
2120058       // Show compilation date and time, plus upper
2120059       // memory limit.
2120060       //
2120061       k_printf ("os32 build %s ram %i Kibyte\n",
2120062                 BUILD_DATE, (int) multiboot.mem_upper);
2120063       //
2120064       // Show also the command line.
2120065       //
2120066       k_printf ("%s\n", multiboot.cmdline);
2120067       //
2120068       // Set 'mb_max', for memory allocation.
2120069       //
2120070       mb_size (multiboot.mem_upper * 1024);
2120071       //
2120072       // keyboard initialization.
2120073       //
2120074       kbd_load ();
2120075       //
2120076       // Block cache initialization.
2120077       //
2120078       blk_cache_init ();
2120079       //
2120080       // PCI bus initialization.
2120081       //
2120082       pci_init ();
2120083       //
2120084       // File system management initialization.
2120085       //
```

```
2120086       fs_init ();
2120087       //
2120088       // Set up processes.
2120089       //
2120090       proc_init ();
2120091       //
2120092       // Set up network.
2120093       //
2120094       net_init ();
2120095     }
2120096   else
2120097     {
2120098       //
2120099       // If it is not a multiboot loader, it is an
2120100       // error.
2120101       //
2120102       k_printf
2120103         ("os32 build %s. ERROR. no "
2120104          "\"multiboot\" header!\n", BUILD_DATE);
2120105       k_printf ("[%s] system halted\n", __func__);
2120106       k_exit ();
2120107     }
2120108   //
2120109   // The kernel will run interactively.
2120110   //
2120111   k_printf
2120112     (".-------------------------------------------------."
2120113      "\n"
2120114      "| 'h' followed by [Enter] to show a menu           |"
2120115      "\n"
2120116      "'-------------------------------------------------'"
2120117      "\n");
2120118   // menu ();
2120119   //
2120120   //
2120121   //
2120122   for (exit = 0; exit == 0;)
2120123     {
2120124       //
2120125       // While in kernel code, timer interrupt don't
2120126       // start the
2120127       // scheduler. The kernel must leave control to
2120128       // the scheduler
2120129       // via a null system call.
2120130       //
2120131       sys (SYS_0, NULL, 0);
2120132       //
2120133       // Back to work: read the keyboard from the TTY
2120134       // device.
2120135       //
2120136       count =
2120137         dev_io ((pid_t) 0, (dev_t) DEV_TTY, DEV_READ,
2120138                 (off_t) 0, command, (size_t) MAX_CANON,
2120139                 NULL);
2120140       //
2120141       // Check if there is a command: the kernel does
2120142       // not
2120143       // go to sleep.
2120144       //
2120145       if (count < 0)
2120146         {
2120147           if (errno == EAGAIN)
2120148             {
2120149               //
2120150               // No command is ready in the buffer
2120151               // keyboard.
2120152               //
2120153               continue;
2120154             }
2120155           else
2120156             {
2120157               k_perror (NULL);
2120158               continue;
2120159             }
2120160         }
2120161       if (count == 0)
2120162         {
2120163           //
2120164           // No command is ready in the buffer
2120165           // keyboard.
2120166           //
2120167           continue;
2120168         }
2120169       if (count == 1)
2120170         {
2120171           //
2120172           // Just [Enter] was pressed.
```

```
2120173           //
2120174           continue;
2120175         }
2120176       if (count > MAX_CANON)
2120177         {
2120178           //
2120179           // Something impossible!
2120180           //
2120181           continue;
2120182         }
2120183       //
2120184       // The last character is the "line delimiter"
2120185       // (new line et al.),
2120186       // or zero in case of EOF. The last character is
2120187       // replaced with
2120188       // zero, so that the command becomes a
2120189       // terminated string.
2120190       //
2120191       command[count - 1] = 0;
2120192       //
2120193       // A command was typed: start to check what it
2120194       // was.
2120195       //
2120196       if (strncmp (command, "1", MAX_CANON) == 0)
2120197         {
2120198           //
2120199           // Kill init.
2120200           //
2120201           s_kill ((pid_t) 0, (pid_t) 1, SIGKILL);
2120202         }
2120203       else if (strncmp (command, "2", MAX_CANON) == 0)
2120204         {
2120205           //
2120206           // Kill proc. 2
2120207           //
2120208           s_kill ((pid_t) 0, (pid_t) 2, SIGTERM);
2120209         }
2120210       else if (strncmp (command, "3", MAX_CANON) == 0)
2120211         {
2120212           //
2120213           // Kill proc. 3
2120214           //
2120215           s_kill ((pid_t) 0, (pid_t) 3, SIGTERM);
2120216         }
2120217       else if (strncmp (command, "4", MAX_CANON) == 0)
2120218         {
2120219           //
2120220           // Kill proc. 4
2120221           //
2120222           s_kill ((pid_t) 0, (pid_t) 4, SIGTERM);
2120223         }
2120224       else if (strncmp (command, "5", MAX_CANON) == 0)
2120225         {
2120226           //
2120227           // Kill proc. 5
2120228           //
2120229           s_kill ((pid_t) 0, (pid_t) 5, SIGTERM);
2120230         }
2120231       else if (strncmp (command, "6", MAX_CANON) == 0)
2120232         {
2120233           //
2120234           // Kill proc. 6
2120235           //
2120236           s_kill ((pid_t) 0, (pid_t) 6, SIGTERM);
2120237         }
2120238       else if (strncmp (command, "7", MAX_CANON) == 0)
2120239         {
2120240           //
2120241           // Kill proc. 7
2120242           //
2120243           s_kill ((pid_t) 0, (pid_t) 7, SIGTERM);
2120244         }
2120245       else if (strncmp (command, "8", MAX_CANON) == 0)
2120246         {
2120247           //
2120248           // Kill proc. 8
2120249           //
2120250           s_kill ((pid_t) 0, (pid_t) 8, SIGTERM);
2120251         }
2120252       else if (strncmp (command, "9", MAX_CANON) == 0)
2120253         {
2120254           //
2120255           // Kill proc. 9
2120256           //
2120257           s_kill ((pid_t) 0, (pid_t) 9, SIGTERM);
2120258         }
2120259       else if (strncmp (command, "A", MAX_CANON) == 0)
```

```
2120260         {
2120261           //
2120262           // Kill proc. 10
2120263           //
2120264           s_kill ((pid_t) 0, (pid_t) 10, SIGTERM);
2120265         }
2120266       else if (strncmp (command, "B", MAX_CANON) == 0)
2120267         {
2120268           //
2120269           // Kill proc. 11
2120270           //
2120271           s_kill ((pid_t) 0, (pid_t) 11, SIGTERM);
2120272         }
2120273       else if (strncmp (command, "C", MAX_CANON) == 0)
2120274         {
2120275           //
2120276           // Kill proc. 12
2120277           //
2120278           s_kill ((pid_t) 0, (pid_t) 12, SIGTERM);
2120279         }
2120280       else if (strncmp (command, "D", MAX_CANON) == 0)
2120281         {
2120282           //
2120283           // Kill proc. 13
2120284           //
2120285           s_kill ((pid_t) 0, (pid_t) 13, SIGTERM);
2120286         }
2120287       else if (strncmp (command, "E", MAX_CANON) == 0)
2120288         {
2120289           //
2120290           // Kill proc. 14
2120291           //
2120292           s_kill ((pid_t) 0, (pid_t) 14, SIGTERM);
2120293         }
2120294       else if (strncmp (command, "F", MAX_CANON) == 0)
2120295         {
2120296           //
2120297           // Kill proc. 15
2120298           //
2120299           s_kill ((pid_t) 0, (pid_t) 15, SIGTERM);
2120300         }
2120301       else if (strncmp (command, "a", MAX_CANON) == 0)
2120302         {
2120303           run ("/bin/aaa", NULL, NULL);
2120304         }
2120305       else if (strncmp (command, "b", MAX_CANON) == 0)
2120306         {
2120307           run ("/bin/bbb", NULL, NULL);
2120308         }
2120309       else if (strncmp (command, "c", MAX_CANON) == 0)
2120310         {
2120311           run ("/bin/ccc", NULL, NULL);
2120312         }
2120313       else if (strncmp (command, "f", MAX_CANON) == 0)
2120314         {
2120315           pid = fork ();
2120316           if (pid == -1)
2120317             {
2120318               k_perror (NULL);
2120319             }
2120320           else if (pid == 0)
2120321             {
2120322               //
2120323               // Get child real pid.
2120324               //
2120325               pid = getpid ();
2120326               //
2120327               // Please note that the child is no more
2120328               // a kernel, and can access to process
2120329               // system calls.
2120330               //
2120331               for (counter = 0; counter < 60; counter++)
2120332                 {
2120333                   z_printf ("%1x", (int) pid);
2120334                   sleep (1);
2120335                 }
2120336             }
2120337           else
2120338             {
2120339               z_printf ("io sono il genitore di %i\n", pid);
2120340             }
2120341         }
2120342       else if (strncmp (command, "g", MAX_CANON) == 0)
2120343         {
2120344           gdt_print (&gdt_register, 0, 20);
2120345         }
2120346       else if (strncmp (command, "G", MAX_CANON) == 0)
```



```
2120347         {
2120348           gdt_print (&gdt_register, 21, 41);
2120349         }
2120350       else if (strncmp (command, "h", MAX_CANON) == 0)
2120351         {
2120352           menu ();
2120353         }
2120354       else if (strncmp (command, "i", MAX_CANON) == 0)
2120355         {
2120356           idt_print (&idt_register, 0, 20);
2120357         }
2120358       else if (strncmp (command, "I", MAX_CANON) == 0)
2120359         {
2120360           idt_print (&idt_register, 21, 41);
2120361         }
2120362       else if (strncmp (command, "m", MAX_CANON) == 0)
2120363         {
2120364           mb_print ();
2120365         }
2120366       else if (strncmp (command, "n", MAX_CANON) == 0)
2120367         {
2120368           inode_print ();
2120369         }
2120370       else if (strncmp (command, "p", MAX_CANON) == 0)
2120371         {
2120372           proc_print ();
2120373         }
2120374       else if (strncmp (command, "s", MAX_CANON) == 0)
2120375         {
2120376           sb_print ();
2120377         }
2120378       else if (strncmp (command, "t", MAX_CANON) == 0)
2120379         {
2120380           k_printf ("clock: %lli time: %i\n",
2120381                     (long long int) k_clock (),
2120382                     (int) k_time (NULL));
2120383         }
2120384       else if (strncmp (command, "T", MAX_CANON) == 0)
2120385         {
2120386           while (1)
2120387             {
2120388               k_printf ("clock: %lli time: %i\n",
2120389                         (long long int) k_clock (),
2120390                         (int) k_time (NULL));
2120391             }
2120392         }
2120393       else if (strncmp (command, "w", MAX_CANON) == 0)
2120394         {
2120395           status = s_open ((pid_t) 0, "/tmp/test",
2120396                            O_WRONLY | O_CREAT |
2120397                            O_TRUNC, 0644);
2120398           //
2120399           if (status >= 0)
2120400             {
2120401               status = s_close ((pid_t) 0, status);
2120402               if (status != 0)
2120403                 {
2120404                   k_perror (NULL);
2120405                 }
2120406             }
2120407         }
2120408       else if (strncmp (command, "x", MAX_CANON) == 0)
2120409         {
2120410           //
2120411           // Load init.
2120412           //
2120413           exec_argv[0] = "/bin/init";
2120414           exec_argv[1] = NULL;
2120415           pid = run ("/bin/init", exec_argv, NULL);
2120416           //
2120417           // Just sleep.
2120418           //
2120419           while (1)
2120420             {
2120421               sys (SYS_0, NULL, 0);
2120422             }
2120423         }
2120424       else if (strncmp (command, "q", MAX_CANON) == 0)
2120425         {
2120426           k_printf ("System halted!\n");
2120427           return;
2120428         }
2120429       else if (strncmp (command, "y", MAX_CANON) == 0)
2120430         {
2120431           // icmp_test3 ();
2120432           // icmp_test2 ();
2120433           // ip_test2 ();
```

```
2120434           // ip_test ();
2120435           tcp_test ();
2120436         }
2120437       else if (strncmp (command, "r", MAX_CANON) == 0)
2120438         {
2120439           arp_print ();
2120440         }
2120441       else if (strncmp (command, "R", MAX_CANON) == 0)
2120442         {
2120443           arp_request (ip_to_be_found);
2120444         }
2120445     }
2120446 }
```

94.9.4 kernel/main/menu.c

Si veda la sezione 93.13.

```
2130001 #include <kernel/main.h>
2130002 #include <kernel/lib_k.h>
2130003 //-------------------------------------------------------
2130004 void
2130005 menu (void)
2130006 {
2130007   k_printf
2130008     (".-------------------------------------------------."
2130009      "\n"
2130010      "| h    show this menu                  .-----------.|"
2130011      "\n"
2130012      "| t    show internal timer values      |all        ||"
2130013      "\n"
2130014      "| f    fork the kernel                 |commands   ||"
2130015      "\n"
2130016      "| m    memory map (HEX)                |followed   ||"
2130017      "\n"
2130018      "| g|G  show GDT table first 21+21 items|by [Enter] ||"
2130019      "\n"
2130020      "| i|I  show IDT table first 21+21 items'----------'|"
2130021      "\n"
2130022      "| p    process status list                         |"
2130023      "\n"
2130024      "| s    super block list                            |"
2130025      "\n"
2130026      "| n    list of active inodes                       |"
2130027      "\n"
2130028      "| 1..9 kill process  1 to 9                        |"
2130029      "\n"
2130030      "| A..F kill process 10 to 15                       |"
2130031      "\n"
2130032      "| a..c run programs '/bin/aaa' to '/bin/ccc'       |"
2130033      "\n"
2130034      "|      in paralel                                  |"
2130035      "\n"
2130036      "| r    ARP table                                   |"
2130037      "\n"
2130038      "| x    exit kernel interaction, start '/bin/init'  |"
2130039      "\n"
2130040      "| q    quit kernel                                 |"
2130041      "\n"
2130042      "'-------------------------------------------------'"
2130043      "\n");
2130044 }
```

94.9.5 kernel/main/run.c

Si veda la sezione 93.13.

```
2140001 #include <kernel/main.h>
2140002 #include <kernel/proc.h>
2140003 #include <kernel/lib_k.h>
2140004 #include <unistd.h>
2140005 //-------------------------------------------------------
2140006 pid_t
2140007 run (char *path, char *argv[], char *envp[])
2140008 {
2140009   pid_t pid;
2140010   //
2140011   pid = fork ();
2140012   if (pid == -1)
2140013     {
2140014       k_perror (NULL);
2140015     }
2140016   else if (pid == 0)
2140017     {
2140018       execve (path, argv, envp);
2140019       z_perror (NULL);
2140020       _exit (0);
2140021     }
2140022   return (pid);
2140023 }
```


94.9.6 kernel/main/stack.s

Si veda la sezione 93.13.

```
2150001 .global _k_stack_top
2150002 .global _k_stack_bottom
2150003 ########################################################
2150004 #
2150005 # Kernel stack size. The value 0x010000 is equal to
2150006 # 1 Mibyte.
2150007 # Please note that if the kernel stack is too little,
2150008 # there is no way to check it.
2150009 #
2150010 .equ STACK_SIZE,  0x0100000
2150011 #-------------------------------------------------------
2150012 .align 4
2150013 .section .bss
2150014 #-------------------------------------------------------
2150015 #
2150016 # At the end is placed the space for the kernel stack,
2150017 # with no initialization.
2150018 #
2150019 _k_stack_top:
2150020 .space STACK_SIZE
2150021 _k_stack_bottom:
2150022 #-------------------------------------------------------
```

94.10 os32: «kernel/memory.h»

Si veda la sezione 93.14.

```
2160001 #ifndef _KERNEL_MEMORY_H
2160002 #define _KERNEL_MEMORY_H       1
2160003 //-------------------------------------------------------
2160004 #include <stdint.h>
2160005 #include <stddef.h>
2160006 #include <sys/types.h>
2160007 //-------------------------------------------------------
2160008 #define MEM_BLOCK_SIZE  0x1000  // 4 Ki/block
2160009 #define MEM_MAX_BLOCKS  0x100000        // 1 Mi blocks
2160010                                 // = 4 Gibyte
2160011
2160012 extern uint32_t mb_table[MEM_MAX_BLOCKS / 32];  // [1]
2160013 //
2160014 // [1] Memory blocks map.
2160015 //
2160016 extern unsigned int mb_max;     // Memory blocks max.
2160017 //-------------------------------------------------------
2160018 typedef unsigned int addr_t;
2160019 //-------------------------------------------------------
2160020 uint32_t *mb_reference (void);
2160021 ssize_t mb_alloc (addr_t address, size_t size);
2160022 void mb_free (addr_t address, size_t size);
2160023 int mb_reduce (addr_t address, size_t new, size_t previous);
2160024 void mb_clean (addr_t address, size_t size);
2160025 addr_t mb_alloc_size (size_t size);
2160026 void mb_print (void);
2160027 void mb_size (size_t size);
2160028 //-------------------------------------------------------
2160029 #endif
```

94.10.1 kernel/memory/mb_alloc.c

Si veda la sezione 93.14.

```
2170001  #include <kernel/memory.h>
2170002  #include <kernel/ibm_i386.h>
2170003  #include <sys/os32.h>
2170004  #include <kernel/lib_k.h>
2170005  //-------------------------------------------------------
2170006  #define DEBUG 0
2170007  //-------------------------------------------------------
2170008  static int mb_block_set1 (int block);
2170009  //-------------------------------------------------------
2170010  ssize_t
2170011  mb_alloc (addr_t address, size_t size)
2170012  {
2170013    unsigned int bstart;
2170014    unsigned int bsize;
2170015    unsigned int bend;
2170016    unsigned int i;
2170017    ssize_t allocated = 0;
2170018    addr_t block_address;
2170019    //
2170020    if (size == 0)
2170021      {
2170022        //
2170023        // Zero means nothing.
2170024        //
2170025        allocated = 0;
2170026        return (allocated);
2170027      }
2170028    //
2170029    // Show what was requested.
2170030    //
2170031    if (DEBUG)
2170032      {
2170033        k_printf ("%s(%i, %zi)", __func__,
2170034                  (unsigned int) address, size);
2170035      }
2170036    //
2170037    // Calculate starting block of memory.
2170038    //
2170039    bstart = address / MEM_BLOCK_SIZE;
2170040    //
2170041    // Calculating size in term of blocks.
2170042    //
2170043    if (size % MEM_BLOCK_SIZE)
2170044      {
2170045        bsize = size / MEM_BLOCK_SIZE + 1;
2170046      }
2170047    else
2170048      {
2170049        bsize = size / MEM_BLOCK_SIZE;
2170050      }
2170051    //
2170052    // Calculate the block number after the
2170053    // end of the requested memory area.
2170054    //
2170055    bend = bstart + bsize;
2170056    //
2170057    // Scan the memory map and allocate.
2170058    //
2170059    for (i = bstart; i < bend; i++)
2170060      {
2170061        if (mb_block_set1 (i))
2170062          {
2170063            allocated += MEM_BLOCK_SIZE;
2170064          }
2170065        else
2170066          {
2170067            block_address = i;
2170068            block_address *= MEM_BLOCK_SIZE;
2170069            k_printf
2170070              ("[%s] Kernel alert: mem block "
2170071               "%x, at address "
2170072               "%x, already allocated!\n", __FILE__, i,
2170073               (unsigned int) block_address);
2170074            break;
2170075          }
2170076      }
2170077    //
2170078    //
2170079    //
2170080    return (allocated);
2170081  }
2170082
2170083  //-------------------------------------------------------
2170084  static int
2170085  mb_block_set1 (int block)
2170086  {
2170087    int i = block / 32;
2170088    int j = block % 32;
2170089    uint32_t mask = 0x80000000 >> j;
2170090    if (mb_table[i] & mask)
2170091      {
2170092        return (0);       // The block is already set to
2170093        // 1 inside the map!
2170094      }
2170095    else
2170096      {
2170097        mb_table[i] = mb_table[i] | mask;
2170098        return (1);
2170099      }
2170100  }
```


94.10.2 kernel/memory/mb_alloc_size.c

Si veda la sezione 93.14.

```
2180001  #include <kernel/memory.h>
2180002  #include <kernel/ibm_i386.h>
2180003  #include <kernel/lib_k.h>
2180004  #include <sys/os32.h>
2180005  #include <errno.h>
2180006  //-------------------------------------------------------
2180007  #define DEBUG 0
2180008  //-------------------------------------------------------
2180009  static int mb_block_status (int block);
2180010  //-------------------------------------------------------
2180011  addr_t
2180012  mb_alloc_size (size_t size)
2180013  {
2180014    unsigned int bsize;
2180015    unsigned int i;
2180016    unsigned int j;
2180017    unsigned int found = 0;
2180018    addr_t alloc_addr;
2180019    ssize_t alloc_size;
2180020    //
2180021    //
2180022    //
2180023    if (size == 0)
2180024      {
2180025        errset (EINVAL);
2180026        return ((addr_t) 0);
2180027      }
2180028    //
2180029    // Show what was requested.
2180030    //
2180031    if (DEBUG)
2180032      {
2180033        k_printf ("%s(%zi)", __func__, size);
2180034      }
2180035    //
2180036    // Calculate block size.
2180037    //
2180038    if (size % MEM_BLOCK_SIZE)
2180039      {
2180040        bsize = size / MEM_BLOCK_SIZE + 1;
2180041      }
2180042    else
2180043      {
2180044        bsize = size / MEM_BLOCK_SIZE;
2180045      }
2180046    //
2180047    // Scan for a contiguous space in memory.
2180048    //
2180049    for (i = 0; i < (mb_max - bsize) && !found; i++)
2180050      {
2180051        for (j = 0; j < bsize; j++)
2180052          {
2180053            found = !mb_block_status (i + j);
2180054            if (!found)
2180055              {
2180056                i += j;
2180057                break;
2180058              }
2180059          }
2180060      }
2180061    //
2180062    // If the space was found, allocate it.
2180063    //
2180064    if (found && (j == bsize))
2180065      {
2180066        alloc_addr = i - 1;
2180067        alloc_addr *= MEM_BLOCK_SIZE;
```

```
2180068       alloc_size = bsize * MEM_BLOCK_SIZE;
2180069       alloc_size =
2180070         mb_alloc (alloc_addr, (size_t) alloc_size);
2180071       //
2180072       if (alloc_size <= 0)
2180073         {
2180074           errset (ENOMEM);
2180075           return ((addr_t) 0);
2180076         }
2180077       else if (alloc_size < size)
2180078         {
2180079           mb_free (alloc_addr, (size_t) alloc_size);
2180080           errset (ENOMEM);
2180081           return ((addr_t) 0);
2180082         }
2180083       else
2180084         {
2180085           //
2180086           // Clean memory before return.
2180087           //
2180088           mb_clean (alloc_addr, (size_t) alloc_size);
2180089           //
2180090           //
2180091           //
2180092           return (alloc_addr);
2180093         }
2180094     }
2180095   else
2180096     {
2180097       errset (ENOMEM);
2180098       return ((addr_t) 0);
2180099     }
2180100 }
2180101
2180102 //-----------------------------------------------------------
2180103 static int
2180104 mb_block_status (int block)
2180105 {
2180106   int i = block / 32;
2180107   int j = block % 32;
2180108   uint32_t mask = 0x80000000 >> j;
2180109   return ((int) (mb_table[i] & mask));
2180110 }
```

94.10.3 kernel/memory/mb_clean.c

«

Si veda la sezione 93.14.

```
2190001 #include <kernel/memory.h>
2190002 //-----------------------------------------------------------
2190003 void
2190004 mb_clean (addr_t address, size_t size)
2190005 {
2190006   unsigned int i;
2190007   char *mem;
2190008   //
2190009   mem = (char *) address;
2190010   //
2190011   for (i = 0; i < size; i++)
2190012     {
2190013       mem[i] = 0;
2190014     }
2190015 }
```

94.10.4 kernel/memory/mb_free.c

«

Si veda la sezione 93.14.

```
2200001 #include <kernel/memory.h>
2200002 #include <kernel/ibm_i386.h>
2200003 #include <sys/os32.h>
2200004 #include <kernel/lib_k.h>
2200005 //-----------------------------------------------------------
2200006 #define DEBUG 0
2200007 //-----------------------------------------------------------
2200008 static int mb_block_set0 (int block);
2200009 //-----------------------------------------------------------
2200010 void
2200011 mb_free (addr_t address, size_t size)
2200012 {
2200013   unsigned int bstart;
2200014   unsigned int bsize;
2200015   unsigned int bend;
2200016   unsigned int i;
2200017   addr_t block_address;
2200018   //
2200019   // k_printf ("releasing 0x%x, size 0x%x\n", (int)
2200020   // address,
2200021   // (int) size);
2200022   //
2200023   if (size == 0)
2200024     {
2200025       //
2200026       // Zero means nothing.
2200027       //
2200028       return;
2200029     }
2200030   //
2200031   // Show what was requested.
2200032   //
2200033   if (DEBUG)
2200034     {
2200035       k_printf ("%s(%i, %zi)", __func__,
2200036                 (unsigned int) address, size);
2200037     }
2200038   //
2200039   // Calculate size in term of blocks.
2200040   //
2200041   if (size % MEM_BLOCK_SIZE)
2200042     {
2200043       bsize = size / MEM_BLOCK_SIZE + 1;
2200044     }
2200045   else
2200046     {
2200047       bsize = size / MEM_BLOCK_SIZE;
2200048     }
2200049   //
2200050   // Calculate start address in term of blocks.
2200051   //
2200052   bstart = address / MEM_BLOCK_SIZE;
2200053   //
2200054   // Calculate end address, in term of blocks.
2200055   // This address is after the memory area to
2200056   // be released.
2200057   //
2200058   bend = bstart + bsize;
2200059   //
2200060   // Scan the memory map to free memory blocks.
2200061   //
2200062   for (i = bstart; i < bend; i++)
2200063     {
2200064       if (mb_block_set0 (i))
2200065         {
2200066           ;
2200067         }
2200068       else
2200069         {
2200070           block_address = i;
2200071           block_address *= MEM_BLOCK_SIZE;
2200072           k_printf
2200073             ("[%s] Kernel alert: mem block "
2200074              "0x%x, at address "
2200075              "0x%x, already released!\n", __FILE__, i,
2200076              (unsigned int) block_address);
2200077         }
2200078     }
2200079 }
2200080
2200081 //-----------------------------------------------------------
2200082 static int
2200083 mb_block_set0 (int block)
2200084 {
2200085   int i = block / 32;
2200086   int j = block % 32;
2200087   uint32_t mask = 0x80000000 >> j;
2200088   if (mb_table[i] & mask)
2200089     {
2200090       mb_table[i] = mb_table[i] & ~mask;
2200091       return (1);
2200092     }
2200093   else
2200094     {
2200095       return (0);       // The block is already set to
2200096       // 0 inside the map!
2200097     }
2200098 }
```


94.10.5 kernel/memory/mb_print.c

«

Si veda la sezione 93.14.

```
2210001 #include <kernel/memory.h>
2210002 #include <kernel/ibm_i386.h>
2210003 #include <sys/os32.h>
```

```
2210004 #include <kernel/lib_k.h>
2210005 #include <kernel/multiboot.h>
2210006 //-----------------------------------------------------------
2210007 void
2210008 mb_print (void)
2210009 {
2210010   unsigned int block;
2210011   unsigned int blocks =
2210012     (multiboot.mem_upper * 1024 / MEM_BLOCK_SIZE);
2210013   int i;
2210014   int j;
2210015   uint32_t mask;
2210016   unsigned int start = 0;
2210017   unsigned int stop = 0;
2210018   unsigned int status = 0;
2210019   //
2210020   k_printf ("Hex mem map, blocks of %x:", MEM_BLOCK_SIZE);
2210021   //
2210022   for (block = 0; block < blocks; block++)
2210023     {
2210024       i = block / 32;
2210025       j = block % 32;
2210026       mask = 0x80000000 >> j;
2210027       if (mb_table[i] & mask)
2210028         {
2210029           //
2210030           // Allocated block
2210031           //
2210032           if (status == 0)
2210033             {
2210034               status = 1;
2210035               start = block;
2210036             }
2210037         }
2210038       else
2210039         {
2210040           //
2210041           // Not allocated block.
2210042           //
2210043           if (status == 1)
2210044             {
2210045               status = 0;
2210046               stop = block;
2210047             }
2210048         }
2210049       //
2210050       //
2210051       //
2210052       if (stop > 0)
2210053         {
2210054           k_printf (" %x-%x", start, stop);
2210055           start = 0;
2210056           stop = 0;
2210057         }
2210058     }
2210059   k_printf ("\n");
2210060 }
```

94.10.6 kernel/memory/mb_public.c

«

Si veda la sezione 93.14.

```
2220001 #include <kernel/memory.h>
2220002 #include <stdint.h>
2220003 //-----------------------------------------------------------
2220004 uint32_t mb_table[MEM_MAX_BLOCKS / 32]; // Memory
2220005                                        // blocks map.
2220006 unsigned int mb_max = MEM_MAX_BLOCKS;   // Memory
2220007                                        // blocks max.
2220008 //-----------------------------------------------------------
```

94.10.7 kernel/memory/mb_reduce.c

«

Si veda la sezione 93.14.

```
2230001 #include <kernel/memory.h>
2230002 #include <kernel/ibm_i386.h>
2230003 #include <sys/os32.h>
2230004 #include <errno.h>
2230005 //-----------------------------------------------------------
2230006 int
2230007 mb_reduce (addr_t address, size_t new, size_t previous)
2230008 {
2230009   addr_t start;
2230010   addr_t end;
2230011   size_t size;
2230012   //
2230013   //
2230014   //
2230015   if (new > previous)
2230016     {
2230017       //
2230018       // We are reducing, not extending!
2230019       //
2230020       errset (EINVAL);
2230021       return (-1);
2230022     }
2230023   //
2230024   //
2230025   //
2230026   if (new == previous)
2230027     {
2230028       //
2230029       // Nothing to do.
2230030       //
2230031       return (0);
2230032     }
2230033   //
2230034   // Correct sizes to conform to memory blocks.
2230035   //
2230036   start = address + new;
2230037   if (start % MEM_BLOCK_SIZE)
2230038     {
2230039       start /= MEM_BLOCK_SIZE;
2230040       start++;
2230041       start *= MEM_BLOCK_SIZE;
2230042     }
2230043   //
2230044   end = address + previous;
2230045   end /= MEM_BLOCK_SIZE;
2230046   end *= MEM_BLOCK_SIZE;
2230047   //
2230048   size = end - start;
2230049   //
2230050   // Finally release the extra memory, no more used.
2230051   //
2230052   mb_free (start, size);
2230053   //
2230054   // Ok.
2230055   //
2230056   return (0);
2230057 }
```

94.10.8 kernel/memory/mb_reference.c

«

Si veda la sezione 93.14.

```
2240001 #include <stdint.h>
2240002 #include <kernel/memory.h>
2240003 //-----------------------------------------------------------
2240004 uint32_t *
2240005 mb_reference (void)
2240006 {
2240007   return mb_table;
2240008 }
```

94.10.9 kernel/memory/mb_size.c

«

Si veda la sezione 93.14.

```
2250001 #include <kernel/memory.h>
2250002 //-----------------------------------------------------------
2250003 void
2250004 mb_size (size_t size)
2250005 {
2250006   mb_max = size / MEM_BLOCK_SIZE;
2250007 }
```

94.11 os32: «kernel/multiboot.h»

«

Si veda la sezione 93.15.

```
2260001 #ifndef _KERNEL_MULTIBOOT_H
2260002 #define _KERNEL_MULTIBOOT_H    1
2260003 //-----------------------------------------------------------
2260004 #include <inttypes.h>
2260005 //-----------------------------------------------------------
2260006 #define MBOOT_CMDLINE_MAX           4096
2260007 #define MBOOT_CMDLINE_OPTION_MAX    1024
2260008 #define MBOOT_CMDLINE_ARGUMENTS_MAX 32
2260009 //-----------------------------------------------------------
2260010 typedef struct
2260011 {
```

```
2260012   uint32_t flags;
2260013   uint32_t mem_lower;
2260014   uint32_t mem_upper;
2260015   uint32_t boot_device;
2260016   char *cmdline;
2260017 } multiboot_t;
2260018 //
2260019 typedef struct
2260020 {
2260021   uint32_t flags;
2260022   uint32_t mem_lower;
2260023   uint32_t mem_upper;
2260024   uint32_t boot_device;
2260025   char cmdline[MBOOT_CMDLINE_MAX];
2260026 } multiboot_save_t;
2260027 //
2260028 extern multiboot_save_t multiboot;
2260029 //-------------------------------------------------------
2260030 void mboot_save (multiboot_t * mboot_data);
2260031 char **mboot_cmdline_opt (const char *opt,
2260032                           const char *delim);
2260033 //-------------------------------------------------------
2260034 #endif
```


94.11.1 kernel/multiboot/mboot_cmdline_opt.c

«

Si veda la sezione 93.15.

```
2270001 #include <stddef.h>
2270002 #include <kernel/multiboot.h>
2270003 #include <kernel/lib_k.h>
2270004 #include <string.h>
2270005 #include <errno.h>
2270006 //-------------------------------------------------------
2270007 char **
2270008 mboot_cmdline_opt (const char *opt, const char *delim)
2270009 {
2270010   static char option[MBOOT_CMDLINE_OPTION_MAX];
2270011   static char *argument[MBOOT_CMDLINE_ARGUMENTS_MAX];
2270012   char *a;
2270013   char *z;
2270014   char *t;
2270015   int i;
2270016   size_t size;
2270017   //
2270018   // Check input.
2270019   //
2270020   if (opt == NULL)
2270021     {
2270022       errset (EINVAL);
2270023       return (NULL);
2270024     }
2270025   //
2270026   // Find the option.
2270027   //
2270028   a = strstr (multiboot.cmdline, opt);
2270029   if (a == NULL)
2270030     {
2270031       return (NULL);
2270032     }
2270033   //
2270034   // Find the end of the option: might be a space or
2270035   // the end of the
2270036   // string.
2270037   //
2270038   z = strpbrk (a, " \t\n");
2270039   //
2270040   // Copy the option inside the static array
2270041   // 'option[]'.
2270042   //
2270043   if (z == NULL)
2270044     {
2270045       strncpy (option, a, MBOOT_CMDLINE_OPTION_MAX - 1);
2270046       option[MBOOT_CMDLINE_OPTION_MAX - 1] = 0;
2270047     }
2270048   else
2270049     {
2270050       size = (uintptr_t) z - (uintptr_t) a;
2270051       strncpy (option, a, size);
2270052       option[size] = 0;
2270053     }
2270054   //
```



```
2270055   // Find the option name, to be saved as the first
2270056   // argument.
2270057   //
2270058   t = strtok (option, "=");
2270059   if (t == NULL)
2270060     {
2270061       errset (EUNKNOWN);
2270062       return (NULL);
2270063     }
2270064   argument[0] = t;
2270065   //
2270066   // If there is no delimiter, replace it with a
2270067   // string containing
2270068   // just a space.
2270069   //
2270070   if (delim == NULL)
2270071     {
2270072       delim = " ";
2270073     }
2270074   //
2270075   for (i = 1; i < MBOOT_CMDLINE_ARGUMENTS_MAX; i++)
2270076     {
2270077       t = strtok (NULL, delim);
2270078       if (t == NULL)
2270079         {
2270080           //
2270081           // The argument will be an empty string,
2270082           // taken from
2270083           // the end of the option string.
2270084           //
2270085           argument[i] =
2270086             &option[MBOOT_CMDLINE_OPTION_MAX - 1];
2270087         }
2270088       else
2270089         {
2270090           argument[i] = t;
2270091         }
2270092     }
2270093   //
2270094   // Return.
2270095   //
2270096   return (argument);
2270097 }
```

94.11.2 kernel/multiboot/mboot_public.c

«

Si veda la sezione 93.15.

```
2280001 #include <kernel/multiboot.h>
2280002 //-------------------------------------------------------
2280003 multiboot_save_t multiboot;
```

94.11.3 kernel/multiboot/mboot_save.c

«

Si veda la sezione 93.15.

```
2290001 #include <kernel/multiboot.h>
2290002 #include <string.h>
2290003 #include <kernel/lib_k.h>
2290004 //-------------------------------------------------------
2290005 void
2290006 mboot_save (multiboot_t * mboot_data)
2290007 {
2290008   multiboot.flags = mboot_data->flags;
2290009   //
2290010   if ((mboot_data->flags & 1) > 0)
2290011     {
2290012       multiboot.mem_lower = mboot_data->mem_lower;
2290013       multiboot.mem_upper = mboot_data->mem_upper;
2290014     }
2290015   if ((mboot_data->flags & 2) > 0)
2290016     {
2290017       multiboot.boot_device = mboot_data->boot_device;
2290018     }
2290019   if ((mboot_data->flags & 4) > 0)
2290020     {
2290021       strncpy (multiboot.cmdline, mboot_data->cmdline,
2290022                MBOOT_CMDLINE_MAX);
2290023     }
2290024   else
2290025     {
2290026       memset (multiboot.cmdline, 0, MBOOT_CMDLINE_MAX);
2290027     }
2290028 }
```

94.12 os32: «kernel/net.h»

Si veda la sezione 93.17.

```
2300001 #ifndef _KERNEL_NET_H
2300002 #define _KERNEL_NET_H    1
2300003 //-------------------------------------------------------
2300004 //
2300005 // WARNING: please remember to limit the max packets
2300006 // size to be transmitted, to a value that ensure no
2300007 // fragmentation at IPv4 level.
2300008 //
2300009 //-------------------------------------------------------
2300010 #include <stdint.h>
2300011 #include <sys/types.h>
2300012 #include <netinet/in.h>
2300013 #include <lib/sys/os32.h>
2300014 #include <netinet/ip.h>
2300015 #include <arpa/inet.h>
2300016 //-------------------------------------------------------
2300017 #define NET_PROT_IP     0x0800
2300018 #define NET_PROT_ARP    0x0806
2300019 #define NET_PROT_RARP   0x8035
2300020 //
2300021 #define NET_ETHERNET_ADDRESS_LENGTH   6
2300022 #define NET_IP_ADDRESS_LENGTH         4
2300023 //
2300024 #define NET_DEV_NULL               0x0000
2300025 #define NET_DEV_LOOP               0x0100
2300026 #define NET_DEV_LOOPBACK           0x0101
2300027 #define NET_DEV_ETH                0x0200
2300028 #define NET_DEV_ETH_NE2K           0x0201
2300029 //
2300030 #define NET_MAX_DEVICES               4
2300031 #define NET_MAX_BUFFERS              64
2300032 //
2300033 #define NET_ETHERNET_MIN_PACKET_SIZE  46
2300034 #define NET_ETHERNET_MAX_PACKET_SIZE  1500
2300035 #define NET_ETHERNET_HEADER_SIZE      14
2300036 #define NET_ETHERNET_MAX_FRAME_SIZE \
2300037     (NET_ETHERNET_MAX_PACKET_SIZE \
2300038      +NET_ETHERNET_HEADER_SIZE)
2300039 //
2300040 #define NET_IP_MIN_HEADER_SIZE  20
2300041 #define NET_IP_MIN_PACKET_SIZE  576
2300042 #define NET_IP_MAX_PACKET_SIZE \
2300043     NET_ETHERNET_MAX_PACKET_SIZE        // [1]
2300044 #define NET_IP_MAX_DATA_SIZE \
2300045     (NET_IP_MAX_PACKET_SIZE \
2300046      -NET_IP_MIN_HEADER_SIZE)
2300047 //
2300048 // [1] The IP max packet size should be 65535, but as
2300049 //     os32 does not accept fragmented packets, the
2300050 //     maximum value depends on the Ethernet packet
2300051 //     size. But this limitation might change, so
2300052 //     there is also the value NET_MTU, that depends
2300053 //     from the real physical network limitations.
2300054 //
2300055 #define NET_MTU  NET_ETHERNET_MAX_PACKET_SIZE
2300056 //-------------------------------------------------------
2300057 //
2300058 // IP unfragmented packet, but it is not known if there
2300059 // are IP options, so it is not possible to define the
2300060 // space after the minimal header.
2300061 //
2300062 typedef union
2300063 {
2300064   uint8_t octet[NET_IP_MAX_PACKET_SIZE];
2300065   struct iphdr header;
2300066 } __attribute__ ((packed)) net_ip_packet_t;
2300067 //
2300068 // Ethernet header.
2300069 //
2300070 typedef struct
2300071 {
2300072   uint8_t dst[6];
2300073   uint8_t src[6];
2300074   uint16_t type;
2300075 } __attribute__ ((packed)) net_ethernet_header_t;
2300076 //
2300077 // Ethernet frame.
2300078 //
2300079 // .-------------------------------------------------.
2300080 // | .                                               |
2300081 // |-------------------------------------------------|
2300082 // | .octet[]                                        |
2300083 // |----------.--------------------------------------|
2300084 // | .header  | .packet                              |
2300085 // |----------+--------------------------------------|
2300086 // | ...      | .packet.octet[]                      |
2300087 // |----------+---------------.----------------------|
2300088 // |          | .packet.header | / / / / // / / / / |
2300089 // '----------'---------------'----------------------'
2300090 //
2300091 typedef union
2300092 {
2300093   //
2300094   uint8_t octet[NET_ETHERNET_MAX_FRAME_SIZE];
2300095   //
2300096   struct
2300097   {
2300098     net_ethernet_header_t header;
2300099     union
2300100     {
2300101       //
2300102       uint8_t octet[NET_IP_MAX_PACKET_SIZE];
2300103       //
2300104       struct iphdr header;
2300105       //
2300106     } __attribute__ ((packed)) packet;
2300107   } __attribute__ ((packed));
2300108   //
2300109 } __attribute__ ((packed)) net_ethernet_frame_t;
2300110 //-------------------------------------------------------
2300111 //
2300112 // Ethernet buffer.
2300113 //
2300114 typedef struct
2300115 {
2300116   clock_t clock;
2300117   size_t size;
2300118   net_ethernet_frame_t frame;
2300119 } __attribute__ ((packed)) net_buffer_eth_t;
2300120 //
2300121 // Loopback buffer.
2300122 //
2300123 typedef struct
2300124 {
2300125   clock_t clock;
2300126   size_t size;
2300127   union
2300128   {
2300129     uint8_t octet[NET_MTU];
2300130     struct iphdr header;
2300131   } packet;
2300132 } __attribute__ ((packed)) net_buffer_lo_t;     // [2]
2300133 //
2300134 // [2] The structure net_ip_packet_t is not used here,
2300135 //     because it refers to unfragmented packets, where
2300136 //     here the max size might be less.
2300137 //
2300138 //-------------------------------------------------------
2300139 //
2300140 // Network interfaces table structure
2300141 //
2300142 typedef struct
2300143 {
2300144   unsigned int type;
2300145   h_addr_t ip;  // IPv4 address in host byte order.
2300146   uint8_t m;    // Short netmask.
2300147   union
2300148   {
2300149     //
2300150     // Ethernet type data:
2300151     //
2300152     struct
2300153     {
2300154       uint8_t mac[6];
2300155       uintptr_t base_io;
2300156       unsigned char irq;
2300157       net_buffer_eth_t buffer[NET_MAX_BUFFERS];
2300158     } ethernet;
2300159     //
2300160     // Loopback type data:
2300161     //
2300162     struct
2300163     {
2300164       net_buffer_lo_t buffer[NET_MAX_BUFFERS];
2300165     } loopback;
2300166   };
2300167 } net_t;
2300168 //
2300169 // [2] The structure net_ip_packet_t is not used here,
2300170 //     because it refers to unfragmented packets, where
2300171 //     here the max size might be less.
```

```
//
extern net_t net_table[NET_MAX_DEVICES];
//-----------------------------------------------------------
int net_rx (void);
int net_tcp (void);
void net_init (void);
int net_index (h_addr_t ip);
int net_index_eth (h_addr_t ip, uint8_t mac[6],
                   uintptr_t io);

net_buffer_eth_t *net_buffer_eth (int n);
net_buffer_lo_t *net_buffer_lo (int n);
void net_print (void);

//void  net_eth_init              (int start);
int net_eth_tx (int dev, void *buffer, size_t size);
int net_eth_ip_tx (h_addr_t src, h_addr_t dst,
                   const void *packet, size_t size);
//-----------------------------------------------------------
#endif
```


94.12.1 kernel/net/arp.h

Si veda la sezione 93.1.

```
#ifndef _KERNEL_NET_ARP_H
#define _KERNEL_NET_ARP_H    1
//-----------------------------------------------------------
#include <stdint.h>
#include <sys/types.h>
#include <kernel/net/ip.h>
#include <arpa/inet.h>
//-----------------------------------------------------------
#define ARP_HW_ETHERNET    1
#define ARP_HW_IEEE802     6    // Example, but not
                                // used.
//-----------------------------------------------------------
#define ARP_TYPE_REQUEST   1
#define ARP_TYPE_REPLY     2
//-----------------------------------------------------------
typedef struct
{
  uint16_t hardware_type;
  uint16_t protocol_type;
  uint8_t hardware_address_length;
  uint8_t protocol_address_length;
  uint16_t opcode;
  uint8_t sender_mac[NET_ETHERNET_ADDRESS_LENGTH];
  uint32_t sender_ip;   // Network byte order.
  uint8_t target_mac[NET_ETHERNET_ADDRESS_LENGTH];
  uint32_t target_ip;   // Network byte order.
  uint8_t filler[18];   // [1]
} __attribute__ ((packed)) arp_packet_t;
//
// [1] The filler is just big enough to get a minimal
//     Ethernet frame size.
//
//-----------------------------------------------------------
#define ARP_MAX_ITEMS 64
#define ARP_MAX_TIME  300       // Seconds.
//
typedef struct
{
  time_t time;
  uint8_t mac[NET_ETHERNET_ADDRESS_LENGTH];
  h_addr_t ip;  // Host byte order.
} arp_t;
//
extern arp_t arp_table[ARP_MAX_ITEMS];
//-----------------------------------------------------------
void arp_clean (void);
int arp_index (unsigned char mac[6], h_addr_t ip);
void arp_init (void);
void arp_print (void);
arp_t *arp_reference (void);
void arp_request (h_addr_t ip);
int arp_rx (int n, int f);
//-----------------------------------------------------------
#endif
```

94.12.2 kernel/net/arp/arp_clean.c

Si veda la sezione 93.1.

```
2320001 #include <kernel/net/arp.h>
2320002 #include <kernel/net/ip.h>
2320003 #include <kernel/lib_k.h>
2320004 #include <string.h>
2320005 //-------------------------------------------------------
2320006 void
2320007 arp_clean (void)
2320008 {
2320009   int a;        // ARP table index.
2320010   //
2320011   time_t time_min = k_time (NULL) - ARP_MAX_TIME;
2320012   //
2320013   //
2320014   //
2320015   for (a = 0; a < ARP_MAX_ITEMS; a++)
2320016     {
2320017       if (arp_table[a].time < time_min)
2320018         {
2320019           //
2320020           // Too old: reset the item to all zeroes.
2320021           //
2320022           memset (&arp_table[a], 0x00,
2320023                   sizeof (arp_table[a]));
2320024         }
2320025     }
2320026 }
```

94.12.3 kernel/net/arp/arp_index.c

Si veda la sezione 93.1.

```
2330001 #include <sys/os32.h>
2330002 #include <kernel/net/arp.h>
2330003 #include <kernel/driver/nic/ne2k.h>
2330004 #include <kernel/driver/pci.h>
2330005 #include <kernel/ibm_i386.h>
2330006 #include <errno.h>
2330007 //-------------------------------------------------------
2330008 extern arp_t arp_table[ARP_MAX_ITEMS];
2330009 //-------------------------------------------------------
2330010 int
2330011 arp_index (unsigned char mac[6], h_addr_t ip)
2330012 {
2330013   //
2330014   int a;
2330015   //
2330016   // By mac address.
2330017   //
2330018   if (mac != NULL)
2330019     {
2330020       for (a = 0; a < ARP_MAX_ITEMS; a++)
2330021         {
2330022           if (arp_table[a].mac[0] == mac[0]
2330023               && arp_table[a].mac[1] == mac[1]
2330024               && arp_table[a].mac[2] == mac[2]
2330025               && arp_table[a].mac[3] == mac[3]
2330026               && arp_table[a].mac[4] == mac[4]
2330027               && arp_table[a].mac[5] == mac[5])
2330028             {
2330029               return (a);
2330030             }
2330031         }
2330032     }
2330033   //
2330034   // By IPv4 address.
2330035   //
2330036   if (ip != 0)
2330037     {
2330038       for (a = 0; a < ARP_MAX_ITEMS; a++)
2330039         {
2330040           if (arp_table[a].ip == ip)
2330041             {
2330042               return (a);
2330043             }
2330044         }
2330045     }
2330046   //
2330047   // Not found!
2330048   //
2330049   errset (ENODEV);
2330050   return (-1);
2330051 }
```


94.12.4 kernel/net/arp/arp_init.c

Si veda la sezione 93.1.

```
2340001 #include <kernel/net/arp.h>
2340002 #include <string.h>
2340003 //-------------------------------------------------------
2340004 void
2340005 arp_init (void)
2340006 {
2340007   memset (arp_table, 0x00, sizeof (arp_table));
2340008 }
```

94.12.5 kernel/net/arp/arp_print.c

Si veda la sezione 93.1.

```
2350001 #include <kernel/net/arp.h>
2350002 #include <kernel/net/ip.h>
2350003 #include <kernel/lib_k.h>
2350004 //-------------------------------------------------------
2350005 void
2350006 arp_print (void)
2350007 {
2350008   int a;        // ARP table index.
2350009   //
2350010   for (a = 0; a < ARP_MAX_ITEMS; a++)
2350011     {
2350012       if (arp_table[a].time > 0)
2350013         {
2350014           k_printf ("%i.%i.%i.%i  ",
2350015                     arp_table[a].ip >> 24 & 0x000000FF,
2350016                     arp_table[a].ip >> 16 & 0x000000FF,
2350017                     arp_table[a].ip >> 8 & 0x000000FF,
2350018                     arp_table[a].ip >> 0 & 0x000000FF);
2350019           //
2350020           k_printf ("%02x:%02x:%02x:%02x:%02x:%02x  ",
2350021                     arp_table[a].mac[0],
2350022                     arp_table[a].mac[1],
2350023                     arp_table[a].mac[2],
2350024                     arp_table[a].mac[3],
2350025                     arp_table[a].mac[4],
2350026                     arp_table[a].mac[5]);
2350027           //
2350028           k_printf ("%3us\n",
2350029                     (unsigned int)
2350030                     (k_time (NULL) - arp_table[a].time));
2350031           //
2350032         }
2350033     }
2350034 }
```

94.12.6 kernel/net/arp/arp_public.c

Si veda la sezione 93.1.

```
2360001 #include <kernel/net/arp.h>
2360002 //-------------------------------------------------------
2360003 arp_t arp_table[ARP_MAX_ITEMS];
2360004 //-------------------------------------------------------
```

94.12.7 kernel/net/arp/arp_reference.c

Si veda la sezione 93.1.

```
2370001 #include <kernel/net/arp.h>
2370002 #include <kernel/net/ip.h>
2370003 #include <sys/os32.h>
2370004 #include <kernel/lib_k.h>
2370005 //-------------------------------------------------------
2370006 #define DEBUG 0
2370007 //-------------------------------------------------------
2370008 arp_t *
2370009 arp_reference (void)
2370010 {
2370011   int a;        // ARP table index.
2370012   time_t older = 0;
2370013   //
2370014   for (a = 0; a < ARP_MAX_ITEMS; a++)
2370015     {
2370016       if (arp_table[a].time == 0)
2370017         {
2370018           //
2370019           // Enough.
2370020           //
2370021           return (&arp_table[a]);
2370022         }
2370023       //
```

```
2370024         older = min (arp_table[a].time, older);
2370025       }
2370026     //
2370027     return (&arp_table[a]);
2370028   }
```

94.12.8 kernel/net/arp/arp_request.c

Si veda la sezione 93.1.

```
2380001 #include <kernel/net.h>
2380002 #include <kernel/net/arp.h>
2380003 #include <kernel/net/ip.h>
2380004 #include <sys/os32.h>
2380005 #include <kernel/lib_k.h>
2380006 #include <errno.h>
2380007 #include <arpa/inet.h>
2380008 //-------------------------------------------------------
2380009 #define DEBUG 0
2380010 //-------------------------------------------------------
2380011 void
2380012 arp_request (h_addr_t ip)
2380013 {
2380014   const uint8_t
2380015     ethernet_broadcast[NET_ETHERNET_ADDRESS_LENGTH] =
2380016     { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF };
2380017   const uint8_t
2380018     ethernet_null[NET_ETHERNET_ADDRESS_LENGTH] =
2380019     { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
2380020   //
2380021   net_ethernet_frame_t frame;
2380022   arp_packet_t *arp = (arp_packet_t *) & frame.packet;
2380023   int n;        // NET table index.
2380024   int i;
2380025   //
2380026   // Send the ARP request to all Ethernet interfaces.
2380027   //
2380028   for (n = 0; n < NET_MAX_DEVICES; n++)
2380029     {
2380030       if (net_table[n].type & NET_DEV_ETH)
2380031         {
2380032           //
2380033           // Build the ARP request packet, starting
2380034           // from Ethernet
2380035           // MAC addresses and protocol type.
2380036           //
2380037           memcpy (frame.header.src,
2380038                   net_table[n].ethernet.mac,
2380039                   (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2380040           memcpy (frame.header.dst, ethernet_broadcast,
2380041                   (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2380042           frame.header.type = htons (NET_PROT_ARP);
2380043           //
2380044           // Now the ARP packet inside.
2380045           //
2380046           arp->hardware_type = htons (ARP_HW_ETHERNET);
2380047           arp->protocol_type = htons (NET_PROT_IP);
2380048           arp->hardware_address_length
2380049             = NET_ETHERNET_ADDRESS_LENGTH;
2380050           arp->protocol_address_length
2380051             = NET_IP_ADDRESS_LENGTH;
2380052           arp->opcode = htons (ARP_TYPE_REQUEST);
2380053           memcpy (arp->sender_mac,
2380054                   net_table[n].ethernet.mac,
2380055                   (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2380056           arp->sender_ip = htonl (net_table[n].ip);
2380057           memcpy (arp->target_mac, ethernet_null,
2380058                   (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2380059           //
2380060           arp->target_ip = htonl (ip);
2380061           //
2380062           for (i = 0; i < (sizeof_array (arp->filler)); i++)
2380063             {
2380064               arp->filler[i] = 0;
2380065             }
2380066           //
2380067           // Send it.
2380068           //
2380069           net_eth_tx (n, &frame,
2380070                       sizeof (arp_packet_t) + 14);
2380071         }
2380072     }
2380073 }
```


94.12.9 kernel/net/arp/arp_rx.c

Si veda la sezione 93.1.

```
2390001 #include <kernel/net.h>
2390002 #include <kernel/net/arp.h>
2390003 #include <kernel/net/ip.h>
2390004 #include <sys/os32.h>
2390005 #include <kernel/lib_k.h>
2390006 #include <errno.h>
2390007 #include <arpa/inet.h>
2390008 //-------------------------------------------------------
2390009 #define DEBUG 0
2390010 //-------------------------------------------------------
2390011 int
2390012 arp_rx (int n, int f)
2390013 {
2390014   net_ethernet_frame_t *frame =
2390015     &net_table[n].ethernet.buffer[f].frame;
2390016   arp_packet_t *arp = (arp_packet_t *) & frame->packet;
2390017   int i;
2390018   int a;        // ARP table index.
2390019   arp_t *arp_table_new_item;
2390020   //
2390021   net_ethernet_frame_t ans_frame;
2390022   arp_packet_t *ans_arp =
2390023     (arp_packet_t *) & ans_frame.packet;
2390024   //
2390025   //
2390026   //
2390027   if (n >= NET_MAX_DEVICES || n < 0)
2390028     {
2390029       errset (EINVAL);  // Invalid argument.
2390030       return (-1);
2390031     }
2390032   //
2390033   if (!(net_table[n].type & NET_DEV_ETH))
2390034     {
2390035       errset (EINVAL);  // Invalid argument.
2390036       return (-1);
2390037     }
2390038   //
2390039   if (ntohs (frame->header.type) != NET_PROT_ARP)
2390040     {
2390041       errset (EINVAL);  // Invalid argument.
2390042       return (-1);
2390043     }
2390044   //
2390045   //
2390046   //
2390047   if (ntohs (arp->opcode) == ARP_TYPE_REQUEST)
2390048     {
2390049       //
2390050       // This is an ARP request: we try to answare if
2390051       // the
2390052       // the IP address is owned.
2390053       //
2390054       if (arp->target_ip == htonl (net_table[n].ip))
2390055         {
2390056           //
2390057           // Found IPv4 address. Prepare an answare.
2390058           //
2390059           memcpy (ans_frame.header.dst,
2390060                   arp->sender_mac,
2390061                   (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2390062           memcpy (ans_frame.header.src,
2390063                   net_table[n].ethernet.mac,
2390064                   (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2390065           ans_frame.header.type = htons (NET_PROT_ARP);
2390066           ans_arp->hardware_type = htons (ARP_HW_ETHERNET);
2390067           ans_arp->protocol_type = htons (NET_PROT_IP);
2390068           ans_arp->hardware_address_length
2390069             = NET_ETHERNET_ADDRESS_LENGTH;
2390070           ans_arp->protocol_address_length
2390071             = NET_IP_ADDRESS_LENGTH;
2390072           ans_arp->opcode = htons (ARP_TYPE_REPLY);
2390073           memcpy (ans_arp->sender_mac,
2390074                   net_table[n].ethernet.mac,
2390075                   (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2390076           ans_arp->sender_ip = htonl (net_table[n].ip);
2390077           memcpy (ans_arp->target_mac, arp->sender_mac,
2390078                   (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2390079           ans_arp->target_ip = arp->sender_ip;
2390080           for (i = 0;
2390081                i < (sizeof_array (ans_arp->filler)); i++)
2390082             {
2390083               ans_arp->filler[i] = 0;
2390084             }
2390085           //
```

```
2390086           // Send it.
2390087           //
2390088           net_eth_tx (n, &ans_frame,
2390089                       sizeof (arp_packet_t) + 14);
2390090         }
2390091       //
2390092       // Done.
2390093       //
2390094       return (0);
2390095     }
2390096   //
2390097   //
2390098   //
2390099   if (ntohs (arp->opcode) == ARP_TYPE_REPLY)
2390100     {
2390101       //
2390102       // We should save it inside the ARP table.
2390103       //
2390104       for (a = 0; a < ARP_MAX_ITEMS; a++)
2390105         {
2390106           //
2390107           // Check if we already have the same item.
2390108           //
2390109           if (memcmp
2390110               (arp->sender_mac, arp_table[a].mac,
2390111                (size_t) NET_ETHERNET_ADDRESS_LENGTH) == 0)
2390112             {
2390113               if (arp_table[a].ip == ntohl (arp->sender_ip))
2390114                 {
2390115                   //
2390116                   // Found: update the time.
2390117                   //
2390118                   arp_table[a].time = k_time (NULL);
2390119                   //
2390120                   // Done.
2390121                   //
2390122                   return (0);
2390123                 }
2390124               else
2390125                 {
2390126                   //
2390127                   // There is already the Ethernet
2390128                   // address,
2390129                   // but the IP is different.
2390130                   //
2390131                   if (DEBUG)
2390132                     {
2390133                       k_printf
2390134                         ("%s:%i: Ethernet address "
2390135                          "%x02:%02:%x02:%x02:%02:%x02 "
2390136                          "has more than one IP\n",
2390137                          __FILE__, __LINE__,
2390138                          arp_table[a].mac[0],
2390139                          arp_table[a].mac[1],
2390140                          arp_table[a].mac[2],
2390141                          arp_table[a].mac[3],
2390142                          arp_table[a].mac[4],
2390143                          arp_table[a].mac[5]);
2390144                       //
2390145                       // End of scan.
2390146                       //
2390147                       break;
2390148                     }
2390149                 }
2390150             }
2390151         }
2390152       //
2390153       // If we are here, the MAC-IP couple is new: get
2390154       // a new ARP item.
2390155       //
2390156       arp_table_new_item = arp_reference ();
2390157       //
2390158       memcpy (arp_table_new_item->mac, arp->sender_mac,
2390159               (size_t) NET_ETHERNET_ADDRESS_LENGTH);
2390160       arp_table_new_item->ip = ntohl (arp->sender_ip);
2390161       arp_table_new_item->time = k_time (NULL);
2390162       //
2390163       // Done.
2390164       //
2390165       return (0);
2390166     }
2390167   //
2390168   // If we are here, we don't know the ARP type.
2390169   //
2390170   if (DEBUG)
2390171     {
2390172       k_printf ("%s:%i: unknown ARP type: %i\n",
2390173                 (int) ntohs (arp->opcode));
2390174     }
2390175   //
2390176   // Done.
2390177   //
2390178   return (0);
2390179 }
```


94.12.10 kernel/net/icmp.h

Si veda la sezione 93.8.

```
2400001 #ifndef _KERNEL_NET_ICMP_H
2400002 #define _KERNEL_NET_ICMP_H    1
2400003 //-------------------------------------------------------
2400004 #include <stdint.h>
2400005 #include <sys/types.h>
2400006 #include <kernel/net/ip.h>
2400007 #include <netinet/icmp.h>
2400008 //-------------------------------------------------------
2400009 #define ICMP_HEADER_SIZE   8
2400010 #define ICMP_MAX_PACKET_SIZE  NET_IP_MAX_DATA_SIZE
2400011 #define ICMP_MAX_DATA_SIZE \
2400012     ICMP_MAX_PACKET_SIZE-ICMP_HEADER_SIZE
2400013 //-------------------------------------------------------
2400014 //
2400015 // ICMP packet, for transmission.
2400016 //
2400017 typedef struct
2400018 {
2400019   struct icmphdr header;
2400020   uint8_t data[ICMP_MAX_DATA_SIZE];
2400021 } __attribute__ ((packed)) icmp_packet_t;
2400022
2400023 //-------------------------------------------------------
2400024 int icmp_rx (int i);
2400025 //
2400026 int icmp_tx (h_addr_t src, h_addr_t dst,
2400027              int type, int code, icmp_packet_t * icmp,
2400028              size_t size);
2400029 //
2400030 int icmp_tx_echo (h_addr_t src, h_addr_t dst,
2400031                   int type, int code,
2400032                   int identifier, int sequence,
2400033                   uint8_t * data, size_t size);
2400034 //
2400035 int icmp_tx_unreachable (h_addr_t src, h_addr_t dst,
2400036                          int type, int code,
2400037                          uint8_t * data, size_t size);
2400038 //
2400039 //-------------------------------------------------------
2400040 #endif
```

94.12.11 kernel/net/icmp/icmp_rx.c

Si veda la sezione 93.8.

```
2410001 #include <sys/os32.h>
2410002 #include <kernel/lib_k.h>
2410003 #include <arpa/inet.h>
2410004 #include <kernel/net.h>
2410005 #include <kernel/net/ip.h>
2410006 #include <kernel/net/icmp.h>
2410007 #include <netinet/icmp.h>
2410008 #include <netinet/udp.h>
2410009 #include <kernel/lib_k.h>
2410010 #include <errno.h>
2410011 //-------------------------------------------------------
2410012 #define DEBUG 0
2410013 //-------------------------------------------------------
2410014 int
2410015 icmp_rx (int i)
2410016 {
2410017   icmp_packet_t *icmp;
2410018   size_t size;
2410019   struct iphdr *ip;
2410020   struct udphdr *ports;
2410021   h_addr_t dest = 0;
2410022   h_port_t port = 0;
2410023   int s;        // Socket table index.
2410024   //
2410025   //
2410026   //
2410027   if ((i >= IP_MAX_PACKETS) || i < 0)
2410028     {
2410029       errset (EINVAL);
2410030       return (-1);
```

```
2410031     }
2410032   //
2410033   // Find the ICMP packet start, and the ICMP packet
2410034   // size (the IP
2410035   // packet size - the IP header size).
2410036   //
2410037   icmp = (icmp_packet_t *) ip_table[i].pdu4;
2410038   size = ntohs (ip_table[i].packet.header.tot_len)
2410039     - (ip_table[i].packet.header.ihl * 4);
2410040   //
2410041   // This function is used by the kernel, to do
2410042   // something automatically,
2410043   // when some ICMP packets arrive. The kernel does
2410044   // not remove the
2410045   // serviced packets, but must remember what it
2410046   // already have seen.
2410047   // If it finds that the packet clock time stamp is
2410048   // the same as
2410049   // the value saved for the kernel, there is nothing
2410050   // more to be done.
2410051   //
2410052   if (ip_table[i].kernel_serviced == ip_table[i].clock)
2410053     {
2410054       return (0);
2410055     }
2410056   //
2410057   //
2410058   //
2410059   if (icmp->header.type == ICMP_ECHO)   // ECHO
2410060     // REQUEST
2410061     {
2410062       size -= sizeof (struct icmphdr);
2410063       //
2410064       // Reply: please note that source and
2410065       // destination IP addresses
2410066       // are now inverted.
2410067       //
2410068       icmp_tx_echo (ntohl
2410069                     (ip_table[i].packet.header.daddr),
2410070                     ntohl (ip_table[i].packet.header.saddr),
2410071                     ICMP_ECHOREPLY, 0,
2410072                     ntohs (icmp->header.un.echo.id),
2410073                     ntohs (icmp->header.un.echo.sequence),
2410074                     icmp->data, size);
2410075       //
2410076       // ICMP echo request are resolved internally,
2410077       // but the packet
2410078       // might be read from the RAW socket too. So it
2410079       // is not removed
2410080       // from the ip_table[].
2410081       //
2410082       ip_table[i].kernel_serviced = ip_table[i].clock;
2410083     }
2410084   else if (icmp->header.type == ICMP_DEST_UNREACH)
2410085     {
2410086       //
2410087       // UNREACHABLE
2410088       //
2410089       //
2410090       // The code (subtype) is not checked.
2410091       // After the ICMP header there is a copy of the
2410092       // original IP
2410093       // header.
2410094       //
2410095       ip = (struct iphdr *)
2410096         (((uint8_t *) icmp) + sizeof (struct icmphdr));
2410097       //
2410098       dest = ntohl (ip->daddr);
2410099       //
2410100       // If the IP protocol is TCP or UDP, there are
2410101       // also ports.
2410102       //
2410103       if (ip->protocol == IPPROTO_TCP
2410104           || ip->protocol == IPPROTO_UDP)
2410105         {
2410106           //
2410107           // After the IP header copy, there is the
2410108           // TCP or UDP header
2410109           // copy. To be able to get the ports, the
2410110           // UDP and TCP headers
2410111           // are the same.
2410112           //
2410113           ports =
2410114             (struct udphdr *) (((uint8_t *) ip) +
2410115                                (ip->ihl * 4));
2410116           //
2410117           port = ntohs (ports->dest);
```



```
2410118           //
2410119         }
2410120       //
2410121       // Set corresponding sockets unreachable.
2410122       //
2410123       for (s = 0; s < SOCK_MAX_SLOTS; s++)
2410124         {
2410125           if (sock_table[s].active)
2410126             {
2410127               if (sock_table[s].raddr == dest)
2410128                 {
2410129                   if (port == 0
2410130                       || port == sock_table[s].rport)
2410131                     {
2410132                       if (icmp->header.code ==
2410133                           ICMP_NET_UNREACH)
2410134                         {
2410135                           sock_table[s].unreach_net = 1;
2410136                         }
2410137                       else if (icmp->header.code ==
2410138                                ICMP_HOST_UNREACH)
2410139                         {
2410140                           sock_table[s].unreach_host = 1;
2410141                         }
2410142                       else if (icmp->header.code ==
2410143                                ICMP_PROT_UNREACH)
2410144                         {
2410145                           sock_table[s].unreach_prot = 1;
2410146                         }
2410147                       else if (icmp->header.code ==
2410148                                ICMP_PORT_UNREACH)
2410149                         {
2410150                           sock_table[s].unreach_port = 1;
2410151                         }
2410152                       else
2410153                         {
2410154                           sock_table[s].unreach_host = 1;
2410155                         }
2410156                     }
2410157                 }
2410158             }
2410159         }
2410160     }
2410161   else if (icmp->header.type == ICMP_ECHOREPLY)
2410162     {
2410163       //
2410164       // ECHO REPLY
2410165       //
2410166       if (DEBUG)
2410167         {
2410168           k_printf ("received an echo reply\n");
2410169         }
2410170     }
2410171   else
2410172     {
2410173       //
2410174       // No other ICMP type is handled at the moment,
2410175       // but keep
2410176       // the packet, because might be read from a RAW
2410177       // ICMP socket.
2410178       //
2410179       ;
2410180     }
2410181   //
2410182   return (0);
2410183 }
```

94.12.12 kernel/net/icmp/icmp_tx.c

Si veda la sezione 93.8.

```
2420001 #include <sys/os32.h>
2420002 #include <kernel/lib_k.h>
2420003 #include <arpa/inet.h>
2420004 #include <kernel/net.h>
2420005 #include <kernel/net/ip.h>
2420006 #include <kernel/net/icmp.h>
2420007 #include <errno.h>
2420008 //-----------------------------------------------------------
2420009 #define DEBUG 0
2420010 //-----------------------------------------------------------
2420011 int
2420012 icmp_tx (h_addr_t src, h_addr_t dst,
2420013          int type, int code, icmp_packet_t * icmp,
2420014          size_t size)
2420015 {
2420016   uint16_t checksum;
```

```
2420017   //
2420018   // Fill the ICMP header.
2420019   //
2420020   icmp->header.type = type;
2420021   icmp->header.code = code;
2420022   icmp->header.checksum = 0;
2420023   //
2420024   // Set the header checksum.
2420025   //
2420026   checksum =
2420027     ~(ip_checksum ((void *) icmp, size, NULL, (size_t) 0));
2420028   icmp->header.checksum = htons (checksum);
2420029   //
2420030   // Send to the lower network level.
2420031   //
2420032   return (ip_tx
2420033           (src, dst, IPPROTO_ICMP, (void *) icmp, size));
2420034 }
```

94.12.13 kernel/net/icmp/icmp_tx_echo.c

Si veda la sezione 93.8.

```
2430001 #include <sys/os32.h>
2430002 #include <kernel/lib_k.h>
2430003 #include <arpa/inet.h>
2430004 #include <kernel/net.h>
2430005 #include <kernel/net/ip.h>
2430006 #include <kernel/net/icmp.h>
2430007 #include <errno.h>
2430008 //-------------------------------------------------------
2430009 #define DEBUG 0
2430010 //-------------------------------------------------------
2430011 int
2430012 icmp_tx_echo (h_addr_t src, h_addr_t dst, int type,
2430013               int code, int identifier, int sequence,
2430014               uint8_t * data, size_t size)
2430015 {
2430016   icmp_packet_t icmp;
2430017   //
2430018   // Check the data size.
2430019   //
2430020   if (size > ICMP_MAX_DATA_SIZE)
2430021     {
2430022       errset (EINVAL);
2430023       return (-1);
2430024     }
2430025   //
2430026   // Prepare the packet.
2430027   //
2430028   icmp.header.un.echo.id = htons (identifier);
2430029   icmp.header.un.echo.sequence = htons (sequence);
2430030   //
2430031   memcpy (icmp.data, data, size);
2430032   //
2430033   // Send to the lower network level.
2430034   //
2430035   return (icmp_tx
2430036           (src, dst, type, code, (void *) &icmp,
2430037            (sizeof (struct icmphdr) + size)));
2430038 }
```

94.12.14 kernel/net/icmp/icmp_tx_unreachable.c

Si veda la sezione 93.8.

```
2440001 #include <sys/os32.h>
2440002 #include <kernel/lib_k.h>
2440003 #include <arpa/inet.h>
2440004 #include <kernel/net.h>
2440005 #include <kernel/net/ip.h>
2440006 #include <kernel/net/icmp.h>
2440007 #include <errno.h>
2440008 //-------------------------------------------------------
2440009 #define DEBUG 0
2440010 //-------------------------------------------------------
2440011 int
2440012 icmp_tx_unreachable (h_addr_t src, h_addr_t dst,
2440013                      int type, int code,
2440014                      uint8_t * data, size_t size)
2440015 {
2440016   icmp_packet_t icmp;
2440017   //
2440018   // Check the data size and reduce if necessary.
2440019   //
2440020   size = min (size, ICMP_MAX_DATA_SIZE);
2440021   //
```



```
2440022   if (size < 8)
2440023     {
2440024       //
2440025       // The ICMP destination unreachable data must
2440026       // contain
2440027       // at least 64 bits of the original packet.
2440028       //
2440029       errset (EINVAL);
2440030       return (-1);
2440031     }
2440032   //
2440033   // Check the type: must be ICMP_DEST_UNREACH.
2440034   //
2440035   if (type != ICMP_DEST_UNREACH)
2440036     {
2440037       errset (EINVAL);
2440038       return (-1);
2440039     }
2440040   //
2440041   // Must reset the «destination unreachable»
2440042   // header.
2440043   //
2440044   memset (&icmp.header.un, 0, (size_t) 4);
2440045   //
2440046   // Copy the data inside the ICMP packet.
2440047   //
2440048   memcpy (icmp.data, data, size);
2440049   //
2440050   // Send to the lower network level.
2440051   //
2440052   return (icmp_tx
2440053           (src, dst, type, code, (void *) &icmp,
2440054            (sizeof (struct icmphdr) + size)));
2440055 }
```

94.12.15 kernel/net/ip.h

Si veda la sezione 93.9.

```
2450001 #ifndef _KERNEL_NET_IP_H
2450002 #define _KERNEL_NET_IP_H    1
2450003 //-------------------------------------------------------
2450004 #include <stdint.h>
2450005 #include <sys/types.h>
2450006 #include <kernel/net.h>
2450007 #include <netinet/ip.h>
2450008 //-------------------------------------------------------
2450009 #define IP_VERSION          4
2450010 #define IP_TTL              64
2450011 //-------------------------------------------------------
2450012 #define IP_MAX_PACKETS      64
2450013 //-------------------------------------------------------
2450014 //
2450015 // IP packet, for transmission (no options here).
2450016 // It is just the same as 'ip_header_t', with the
2450017 // 'data[]' member addition.
2450018 //
2450019 typedef struct
2450020 {
2450021   struct iphdr header;
2450022   uint8_t data[NET_IP_MAX_DATA_SIZE];
2450023 } __attribute__ ((packed)) ip_packet_t;
2450024 //
2450025 // IP table for IP packets.
2450026 //
2450027 typedef struct
2450028 {
2450029   clock_t clock;        // [1]
2450030   clock_t kernel_serviced;      // [2]
2450031   uint8_t *pdu4;
2450032   union
2450033   {
2450034     uint8_t octet[NET_IP_MAX_PACKET_SIZE];
2450035     struct iphdr header;
2450036   } packet;
2450037 } ip_t;
2450038 //
2450039 // [1] This is the arrival packet clock time stamp.
2450040 //     When a new packet is to be saved inside the
2450041 //     ip_table[], the older packet is replaced, even
2450042 //     if it was not used. No packets are removed,
2450043 //     because they might be read from a RAW socket,
2450044 //     for some reason.
2450045 //
2450046 // [2] This is only for kernel usage, when a connection
2450047 //     is established by the kernel, without a socket.
2450048 //     The member kernel_serviced is used to remember
```

```
//     to have see a certain packet and that the kernel
//     does not have to try to service it again. For
//     example, when an ECHO REQUEST packet arrive, the
//     kernel answares with an ECHO REPLY, but does not
//     remove the packet that might be read from a true
//     RAW socket. So, the kernel writes inside
//     kernel_serviced the same value found inside
//     clock, so that it know that it was already read.
//
//
// External IP table data.
//
extern ip_t ip_table[IP_MAX_PACKETS];
//-----------------------------------------------------------
uint16_t ip_checksum (uint16_t * data1, size_t size1,
                      uint16_t * data2, size_t size2);
int ip_rx (int n, int f);
ip_t *ip_reference (void);
ssize_t ip_header (h_addr_t src, h_addr_t dst,
                   uint16_t id, uint8_t ttl,
                   uint8_t protocol, void *buffer,
                   size_t length);
int ip_tx (h_addr_t src, h_addr_t dst, int protocol,
           const void *buffer, size_t size);
h_addr_t ip_mask (int m);
//-----------------------------------------------------------
#endif
```

94.12.16 kernel/net/ip/ip_checksum.c

Si veda la sezione 93.9.

```
#include <stdint.h>
#include <arpa/inet.h>
#include <kernel/net/ip.h>
//-----------------------------------------------------------
uint16_t
ip_checksum (uint16_t * data1, size_t size1,
             uint16_t * data2, size_t size2)
{
  int i;
  uint32_t sum;
  uint16_t carry;
  uint16_t checksum;
  uint16_t last;
  uint8_t *octet;
  //
  // 2's complement sum.
  //
  sum = 0;
  //
  if (data1 != NULL)
    {
      for (i = 0; i < (size1 / 2); i++)
        {
          sum += ntohs (data1[i]);
        }
      //
      if (size1 % 2)
        {
          //
          // The size is odd, and the last octet must
          // be accounted too.
          //
          octet = (uint8_t *) data1;
          last = octet[size1 - 1];
          last = last << 8;
          sum += last;
        }
    }
  if (data2 != NULL)
    {
      for (i = 0; i < (size2 / 2); i++)
        {
          sum += ntohs (data2[i]);
        }
      //
      if (size2 % 2)
        {
          //
          // The size is odd, and the last octet must
          // be accounted too.
          //
          octet = (uint8_t *) data2;
          last = octet[size2 - 1];
          last = last << 8;
          sum += last;
        }
    }
  //
  // Extract the carries and make the checksum.
  //
  carry = sum >> 16;
  checksum = sum & 0x0000FFFF;
  checksum += carry;
  //
  // End of job.
  //
  return (checksum);
}
```


94.12.17 kernel/net/ip/ip_header.c

Si veda la sezione 93.9.

```
#include <kernel/net.h>
#include <kernel/net/ip.h>
#include <sys/os32.h>
#include <kernel/lib_k.h>
#include <errno.h>
#include <arpa/inet.h>
#include <netinet/ip.h>
//-----------------------------------------------------------
#define DEBUG 0
//-----------------------------------------------------------
ssize_t
ip_header (h_addr_t src, h_addr_t dst, uint16_t id,
           uint8_t ttl, uint8_t protocol, void *buffer,
           size_t length)
{
  struct iphdr header;
  uint16_t checksum;
  //
  // Check size.
  //
  if (length < sizeof (struct iphdr))
    {
      errset (EINVAL);
      return ((ssize_t) - 1);
    }
  //
  // Prepare the header.
  //
  header.version = IP_VERSION;
  header.ihl = sizeof (struct iphdr) / 4;
  header.tos = 0x00;    // Routine, normal.
  header.tot_len = htons (length);
  header.id = htons (id);
  header.frag_off = htons (0x4000);     // Do not
  // fragment!
  header.ttl = ttl;
  header.protocol = protocol;
  header.check = 0;
  header.saddr = htonl (src);
  header.daddr = htonl (dst);
  //
  // Fix the length.
  //
  length = sizeof (struct iphdr);
  //
  // Now set the header checksum.
  //
  checksum = ~(ip_checksum ((void *) &header, length,
                            NULL, (size_t) 0));
  header.check = htons (checksum);
  //
  // Copy the header to the buffer
  //
  memcpy (buffer, &header, length);
  //
  // Return size written.
  //
  return (length);
}
```

94.12.18 kernel/net/ip/ip_mask.c

Si veda la sezione 93.9.

```
#include <kernel/net.h>
#include <kernel/net/route.h>
#include <kernel/net/arp.h>
//-----------------------------------------------------------
h_addr_t
```

```
2480006 ip_mask (int m)
2480007 {
2480008   int i;
2480009   uint32_t mm = 0x80000000;
2480010   h_addr_t mask = 0;
2480011   //
2480012   for (i = 0; i < m; i++)
2480013     {
2480014       mask |= mm;
2480015       mm = mm >> 1;
2480016     }
2480017   //
2480018   return mask;
2480019 }
```

94.12.19 kernel/net/ip/ip_public.c

«

Si veda la sezione 93.9.

```
2490001 #include <kernel/net/ip.h>
2490002 //-------------------------------------------------------
2490003 ip_t ip_table[IP_MAX_PACKETS];
2490004 //-------------------------------------------------------
```

94.12.20 kernel/net/ip/ip_reference.c

«

Si veda la sezione 93.9.

```
2500001 #include <kernel/net.h>
2500002 #include <kernel/net/arp.h>
2500003 #include <kernel/lib_k.h>
2500004 #include <kernel/net/ip.h>
2500005 #include <sys/os32.h>
2500006 //-------------------------------------------------------
2500007 ip_t *
2500008 ip_reference (void)
2500009 {
2500010   int i;        // IP table index.
2500011   clock_t older;
2500012   int o;        // Older IP table index.
2500013   //
2500014   older = ip_table[0].clock;
2500015   o = 0;
2500016   //
2500017   for (i = 0; i < IP_MAX_PACKETS; i++)
2500018     {
2500019       if (ip_table[i].clock == 0)
2500020         {
2500021           //
2500022           // Enough.
2500023           //
2500024           return (&ip_table[i]);
2500025         }
2500026       //
2500027       if (ip_table[i].clock < older)
2500028         {
2500029           older = ip_table[i].clock;
2500030           o = i;
2500031         }
2500032     }
2500033   //
2500034   return (&ip_table[o]);
2500035 }
```

94.12.21 kernel/net/ip/ip_rx.c

«

Si veda la sezione 93.9.

```
2510001 #include <kernel/net.h>
2510002 #include <kernel/net/arp.h>
2510003 #include <kernel/net/icmp.h>
2510004 #include <kernel/net/ip.h>
2510005 #include <sys/os32.h>
2510006 #include <kernel/lib_k.h>
2510007 #include <errno.h>
2510008 #include <arpa/inet.h>
2510009 #include <netinet/udp.h>
2510010 //-------------------------------------------------------
2510011 #define DEBUG 0
2510012 //-------------------------------------------------------
2510013 int
2510014 ip_rx (int n, int f)
2510015 {
2510016   struct iphdr *header;
2510017   net_ip_packet_t *packet;
2510018   uint16_t checksum;
2510019   size_t size_header;
2510020   ip_t *ip_table_item;
2510021   struct udphdr *udp;
2510022   int i;
2510023   int j;        // Net table index.
2510024   int s;        // Socket table index.
2510025   //
2510026   //
2510027   //
2510028   if (n >= NET_MAX_DEVICES || n < 0)
2510029     {
2510030       errset (EINVAL);  // Invalid argument.
2510031       return (-1);
2510032     }
2510033   if (f >= NET_MAX_BUFFERS || f < 0)
2510034     {
2510035       errset (EINVAL);  // Invalid argument.
2510036       return (-1);
2510037     }
2510038   //
2510039   // Get the packet link.
2510040   //
2510041   if (net_table[n].type & NET_DEV_LOOP)
2510042     {
2510043       packet = (net_ip_packet_t *)
2510044         & net_table[n].loopback.buffer[f].packet;
2510045     }
2510046   else if (net_table[n].type & NET_DEV_ETH)
2510047     {
2510048       //
2510049       // It is Ethernet, but must also have an IP
2510050       // packet inside!
2510051       //
2510052       if (ntohs
2510053           (net_table[n].ethernet.buffer[f].frame.
2510054            header.type) != NET_PROT_IP)
2510055         {
2510056           errset (EINVAL);      // Invalid argument.
2510057           return (-1);
2510058         }
2510059       packet = (net_ip_packet_t *)
2510060         & net_table[n].ethernet.buffer[f].frame.packet;
2510061     }
2510062   else
2510063     {
2510064       errset (EINVAL);  // Invalid argument.
2510065       return (-1);
2510066     }
2510067   //
2510068   // The beginning of the packet contains the IP
2510069   // header.
2510070   //
2510071   header = (struct iphdr *) packet;
2510072   //
2510073   // Verify IP header checksum: it is also calculated
2510074   // the real header
2510075   // size.
2510076   //
2510077   size_header = header->ihl * 4;
2510078   checksum =
2510079     ip_checksum ((uint16_t *) header, size_header,
2510080                  NULL, (size_t) 0);
2510081   if (checksum == 0xFFFF || checksum == 0x0000)
2510082     {
2510083       ; // k_printf ("checksum ok\n");
2510084     }
2510085   else
2510086     {
2510087       k_printf ("BAD CHECKSUM: %04x\n", checksum);
2510088       return (0);
2510089     }
2510090   //
2510091   // Is it a fragment? As we are not able to manage
2510092   // fragments,
2510093   // we just check that it is all zero, ignoring the
2510094   // bit 'DF'.
2510095   // That is why we use a mask 0xBFFF.
2510096   //
2510097   if ((ntohs (header->frag_off) & 0xBFFF) != 0)
2510098     {
2510099       //
2510100       // Sorry, we don't manage fragments.
2510101       //
2510102       k_printf
2510103         ("Sorry: we don't manage IP fragments: %04x\n",
2510104          (ntohs (header->frag_off)));
2510105       return (0);
2510106     }
```

```
2510107   else
2510108     {
2510109       //
2510110       // Find a place inside the IP table.
2510111       //
2510112       ip_table_item = ip_reference ();
2510113       //
2510114       // Copy the packet inside the ip_table[] item,
2510115       // then update
2510116       // the link to the data start inside the packet
2510117       // and the
2510118       // clock_t timestamp.
2510119       //
2510120       memcpy (ip_table_item->packet.octet, packet,
2510121               ntohs (header->tot_len));
2510122       ip_table_item->pdu4 = ip_table_item->packet.octet;
2510123       ip_table_item->pdu4 += (header->ihl * 4);
2510124       ip_table_item->clock = k_clock ();
2510125     }
2510126   //
2510127   // Check for destination unreachable, scanning the
2510128   // interface table,
2510129   // to see if there is such address here.
2510130   //
2510131   for (j = 0; j < NET_MAX_DEVICES; j++)
2510132     {
2510133       if (net_table[j].ip == ntohl (header->daddr))
2510134         {
2510135           //
2510136           // Found a valid local address.
2510137           //
2510138           break;
2510139         }
2510140     }
2510141   if (j >= NET_MAX_DEVICES)
2510142     {
2510143       //
2510144       // Local address not found: host unreachable,
2510145       // but the packet
2510146       // is taken anyway.
2510147       //
2510148       icmp_tx_unreachable (ntohl (header->daddr),
2510149                            ntohl (header->saddr),
2510150                            ICMP_DEST_UNREACH,
2510151                            ICMP_HOST_UNREACH,
2510152                            packet->octet,
2510153                            ntohs (header->tot_len));
2510154     }
2510155   //
2510156   // Check for port unreachable, scanning the socket
2510157   // table.
2510158   //
2510159   if (header->protocol == IPPROTO_UDP
2510160       || header->protocol == IPPROTO_TCP)
2510161     {
2510162       //
2510163       // There are ports.
2510164       //
2510165       udp =
2510166         (struct udphdr *) &(packet->octet[header->ihl * 4]);
2510167       //
2510168       for (s = 0; s < SOCK_MAX_SLOTS; s++)
2510169         {
2510170           if (sock_table[s].active
2510171               && sock_table[s].lport == ntohs (udp->dest))
2510172             {
2510173               //
2510174               // Found a matching local port.
2510175               //
2510176               break;
2510177             }
2510178         }
2510179       if (s >= SOCK_MAX_SLOTS)
2510180         {
2510181           //
2510182           // Local port not found: port unreachable,
2510183           // but the packet
2510184           // is taken anyway.
2510185           //
2510186           icmp_tx_unreachable (ntohl (header->daddr),
2510187                                ntohl (header->saddr),
2510188                                ICMP_DEST_UNREACH,
2510189                                ICMP_PORT_UNREACH,
2510190                                packet->octet,
2510191                                ntohs (header->tot_len));
2510192         }
2510193     }
```



```
2510194   //
2510195   // Now do something with the data inside the
2510196   // 'ip_table[]'.
2510197   //
2510198   for (i = 0; i < IP_MAX_PACKETS; i++)
2510199     {
2510200       if (ip_table[i].clock != 0)
2510201         {
2510202           //
2510203           if (ip_table[i].kernel_serviced !=
2510204               ip_table[i].clock
2510205               && ip_table[i].packet.header.protocol ==
2510206               IPPROTO_ICMP)
2510207             {
2510208               icmp_rx (i);
2510209             }
2510210           else
2510211             {
2510212               //
2510213               // At the moment, no other IP protocol
2510214               // managed internally.
2510215               //
2510216               ip_table[i].kernel_serviced =
2510217                 ip_table[i].clock;
2510218             }
2510219         }
2510220     }
2510221   //
2510222   //
2510223   //
2510224   return (0);
2510225 }
```

94.12.22 kernel/net/ip/ip_tx.c

Si veda la sezione 93.9.

```
2520001 #include <kernel/net.h>
2520002 #include <kernel/net/ip.h>
2520003 #include <kernel/net/route.h>
2520004 #include <sys/os32.h>
2520005 #include <kernel/lib_k.h>
2520006 #include <errno.h>
2520007 #include <arpa/inet.h>
2520008 //-------------------------------------------------------
2520009 #define DEBUG 0
2520010 //-------------------------------------------------------
2520011 int
2520012 ip_tx (h_addr_t src, h_addr_t dst, int protocol,
2520013        const void *buffer, size_t size)
2520014 {
2520015   static int id = 0;
2520016   ip_packet_t packet;
2520017   uint16_t checksum;
2520018   int s;        // source net interface.
2520019   int d;        // destination net interface.
2520020   net_buffer_lo_t *loopback;
2520021   //
2520022   // Verify to have a source address.
2520023   //
2520024   if (src == 0)
2520025     {
2520026       //
2520027       // Default source address: get the source
2520028       // address from the routing
2520029       // table, based on the destination.
2520030       //
2520031       src = route_remote_to_local (dst);
2520032       if (src == ((h_addr_t) - 1))
2520033         {
2520034           errset (errno);
2520035           return (-1);
2520036         }
2520037     }
2520038   //
2520039   // Prepare the packet.
2520040   //
2520041   packet.header.version = IP_VERSION;
2520042   packet.header.ihl = sizeof (struct iphdr) / 4;
2520043   packet.header.tos = 0x0000;   // Routine, normal.
2520044   packet.header.tot_len =
2520045     htons (sizeof (struct iphdr) + size);
2520046   packet.header.id = htons (id++);
2520047   //
2520048   // Do not fragment:
2520049   //
2520050   packet.header.frag_off = htons (0x4000);
```

```
2520051     //
2520052     packet.header.ttl = IP_TTL;
2520053     packet.header.protocol = protocol;
2520054     packet.header.check = 0;
2520055     packet.header.saddr = htonl (src);
2520056     packet.header.daddr = htonl (dst);
2520057     //
2520058     // Now set the header checksum.
2520059     //
2520060     checksum =
2520061       ~(ip_checksum
2520062         ((void *) &packet, sizeof (struct iphdr), NULL,
2520063          (size_t) 0));
2520064     packet.header.check = htons (checksum);
2520065     //
2520066     memcpy (packet.data, buffer, size);
2520067     //
2520068     // ////////////////////////////////////////////////
2520069     // Enter here the lower network level.
2520070     // ////////////////////////////////////////////////
2520071     //
2520072     // The new size includes now the IPv4 header
2520073     //
2520074     size = (sizeof (struct iphdr) + size);
2520075     //
2520076     // Check for PDU size.
2520077     //
2520078     if (size > NET_MTU)
2520079       {
2520080         errset (E_PDU_TOO_BIG);
2520081         return (-1);
2520082       }
2520083     //
2520084     // Find the sender interface.
2520085     //
2520086     s = net_index (src);
2520087     if (s < 0)
2520088       {
2520089         errset (errno);   // ENODEV.
2520090         return (-1);
2520091       }
2520092     //
2520093     // Check if the destination is a local interface.
2520094     //
2520095     d = net_index (dst);
2520096     if (d >= 0)
2520097       {
2520098         //
2520099         // It is a local interface, so must change the
2520100         // destination
2520101         // to the loopback device: it must be 'net0'.
2520102         //
2520103         d = 0;
2520104       }
2520105     else
2520106       {
2520107         //
2520108         // Should not be necessary, but for coherence
2520109         // with the rest
2520110         // of the code...
2520111         //
2520112         d = s;
2520113       }
2520114     //
2520115     // Check if the destination is the loopback
2520116     // interface.
2520117     //
2520118     if (net_table[d].type & NET_DEV_LOOP)
2520119       {
2520120         loopback = net_buffer_lo (d);
2520121         if (loopback == NULL)
2520122           {
2520123             errset (errno);
2520124             return (-1);
2520125           }
2520126         loopback->clock = k_clock ();
2520127         loopback->size = size;
2520128         memcpy (&loopback->packet, (void *) &packet, size);
2520129         return (0);
2520130       }
2520131     //
2520132     // The destination wasn't the loopback interface, so
2520133     // check if the
2520134     // source is an Ethernet device.
2520135     //
2520136     if (net_table[s].type & NET_DEV_ETH)
2520137       {
```



```
2520138         //
2520139         // For Ethernet devices another function is
2520140         // responsible
2520141         // for sending the packet.
2520142         //
2520143         return (net_eth_ip_tx
2520144                 (src, dst, (void *) &packet, size));
2520145       }
2520146     //
2520147     // Should never reach the end, but who knows...
2520148     //
2520149     errset (errno);       // ENODEV.
2520150     return (-1);
2520151   }
```

94.12.23 kernel/net/net_buffer_eth.c

Si veda la sezione 93.17.

```
2530001   #include <sys/os32.h>
2530002   #include <kernel/driver/nic/ne2k.h>
2530003   #include <kernel/driver/pci.h>
2530004   #include <kernel/ibm_i386.h>
2530005   #include <errno.h>
2530006   //-------------------------------------------------------
2530007   net_buffer_eth_t *
2530008   net_buffer_eth (int n)
2530009   {
2530010     int b;        // Buffer index.
2530011     int ref = -1; // Reference index.
2530012     clock_t clock = k_clock ();   // Reference clock
2530013     // value.
2530014     //
2530015     // Check Ethernet index.
2530016     //
2530017     if ((n > NET_MAX_DEVICES) || (n < 0))
2530018       {
2530019         errset (EINVAL);
2530020         return (NULL);
2530021       }
2530022     //
2530023     if (!(net_table[n].type & NET_DEV_ETH))
2530024       {
2530025         errset (EINVAL);
2530026         return (NULL);
2530027       }
2530028     //
2530029     // Ethernet found.
2530030     //
2530031     for (b = 0; b < NET_MAX_BUFFERS; b++)
2530032       {
2530033         if (net_table[n].ethernet.buffer[b].clock == 0)
2530034           {
2530035             //
2530036             // Enough.
2530037             //
2530038             return &net_table[n].ethernet.buffer[b];
2530039           }
2530040         else if (net_table[n].ethernet.buffer[b].clock <
2530041                  clock)
2530042           {
2530043             clock = net_table[n].ethernet.buffer[b].clock;
2530044             ref = b;
2530045           }
2530046       }
2530047     //
2530048     // Return the selected frame structure.
2530049     //
2530050     return &net_table[n].ethernet.buffer[ref];
2530051     //
2530052     // Device not found!
2530053     //
2530054     errset (ENODEV);
2530055     return (NULL);
2530056   }
```

94.12.24 kernel/net/net_buffer_lo.c

Si veda la sezione 93.17.

```
2540001   #include <sys/os32.h>
2540002   #include <kernel/net.h>
2540003   #include <kernel/driver/nic/ne2k.h>
2540004   #include <kernel/driver/pci.h>
2540005   #include <kernel/ibm_i386.h>
2540006   #include <errno.h>
2540007   //-------------------------------------------------------
```

```
net_buffer_lo_t *
net_buffer_lo (int n)
{
  int b;        // Buffer index.
  int ref = -1; // Reference index.
  clock_t clock = k_clock ();   // Reference clock
  // value.
  //
  // Check NET table index.
  //
  if ((n > NET_MAX_DEVICES) || (n < 0))
    {
      errset (EINVAL);
      return (NULL);
    }
  //
  if (!(net_table[n].type & NET_DEV_LOOP))
    {
      errset (EINVAL);
      return (NULL);
    }
  //
  // Loopback found.
  //
  for (b = 0; b < NET_MAX_BUFFERS; b++)
    {
      if (net_table[n].loopback.buffer[b].clock == 0)
        {
          //
          // Enough.
          //
          return &net_table[n].loopback.buffer[b];
        }
      else if (net_table[n].loopback.buffer[b].clock <
               clock)
        {
          clock = net_table[n].loopback.buffer[b].clock;
          ref = b;
        }
    }
  //
  // Return the selected frame structure.
  //
  return &net_table[n].loopback.buffer[ref];
  //
  // Device not found!
  //
  errset (ENODEV);
  return (NULL);
}
```

94.12.25 kernel/net/net_eth_ip_tx.c

« Si veda la sezione 93.17.

```
#include <sys/os32.h>
#include <kernel/net/route.h>
#include <kernel/net/ip.h>
#include <kernel/net/arp.h>
#include <kernel/driver/nic/ne2k.h>
#include <kernel/driver/pci.h>
#include <kernel/ibm_i386.h>
#include <errno.h>
//-----------------------------------------------------------
int
net_eth_ip_tx (h_addr_t src, h_addr_t dst,
               const void *packet, size_t size)
{
  net_ethernet_frame_t frame;
  int n;        // NET table index.
  int a;        // ARP table index.
  int i;
  h_addr_t router;
  //
  // Check for PDU size.
  //
  if (size > NET_ETHERNET_MAX_PACKET_SIZE)
    {
      errset (E_PDU_TOO_BIG);
      return (-1);
    }
  //
  // Find the sender interface address.
  //
  n = net_index_eth (src, NULL, (uintptr_t) 0);
  if (n < 0)
    {
      errset (errno);   // ENODEV.
      return (-1);
    }
  //
  // Copy the Ethernet source address into the
  // Ethernet frame
  // header.
  //
  memcpy (frame.header.src, net_table[n].ethernet.mac,
          NET_ETHERNET_ADDRESS_LENGTH);
  //
  // Find if we need a router.
  //
  router = route_remote_to_router (dst);
  //
  if (router != 0 && router != ((h_addr_t) - 1))
    {
      //
      // We need to find the router destination MAC
      // address.
      //
      a = arp_index (NULL, router);
      if (a < 0)
        {
          //
          // There is not the item inside the ARP
          // table. Send a request
          // and return.
          //
          arp_request (router);
          errset (E_ARP_MISSING);
          return (-1);
        }
    }
  else
    {
      //
      // The destination is inside the local network.
      // Find the destination Ethernet address.
      //
      a = arp_index (NULL, dst);
      if (a < 0)
        {
          //
          // There is not the item inside the ARP
          // table. Send a request
          // and return.
          //
          arp_request (dst);
          errset (E_ARP_MISSING);
          return (-1);
        }
    }
  //
  // Copy the Ethernet destination address into the
  // Ethernet frame
  // header: might be the real destination interface,
  // or the
  // router.
  //
  memcpy (frame.header.dst, arp_table[a].mac,
          NET_ETHERNET_ADDRESS_LENGTH);
  //
  // Set the frame type.
  //
  frame.header.type = htons (NET_PROT_IP);
  //
  // Copy the IP packet.
  //
  memcpy (&frame.packet, packet, size);
  //
  // Fill if the size is too little.
  //
  for (i = size; i < NET_ETHERNET_MIN_PACKET_SIZE; i++)
    {
      frame.packet.octet[i] = 0;
    }
  //
  size = max (size, NET_ETHERNET_MIN_PACKET_SIZE);
  //
  // Now, send the Ethernet frame. Index 'n' is the
  // network
  // device number.
  //
  return (net_eth_tx
          (n, &frame, size + NET_ETHERNET_HEADER_SIZE));
}
```

94.12.26 kernel/net/net_eth_tx.c

Si veda la sezione 93.17.

```
2560001 #include <sys/os32.h>
2560002 #include <kernel/net.h>
2560003 #include <kernel/driver/nic/ne2k.h>
2560004 #include <kernel/driver/pci.h>
2560005 #include <kernel/ibm_i386.h>
2560006 #include <errno.h>
2560007 //-------------------------------------------------------
2560008 int
2560009 net_eth_tx (int n, void *buffer, size_t size)
2560010 {
2560011   //
2560012   if (n >= NET_MAX_DEVICES || n < 0)
2560013     {
2560014       errset (EINVAL);
2560015       return (-1);
2560016     }
2560017   if (!(net_table[n].type & NET_DEV_ETH))
2560018     {
2560019       errset (EINVAL);
2560020       return (-1);
2560021     }
2560022   //
2560023   if (net_table[n].type == NET_DEV_ETH_NE2K)
2560024     {
2560025       return (ne2k_tx
2560026               (net_table[n].ethernet.base_io, buffer,
2560027                size));
2560028     }
2560029   //
2560030   // If we are here, there is not the driver for the
2560031   // Ethernet device.
2560032   //
2560033   errset (ENODEV);
2560034   return (-1);
2560035 }
```

94.12.27 kernel/net/net_index.c

Si veda la sezione 93.17.

```
2570001 #include <sys/os32.h>
2570002 #include <kernel/net.h>
2570003 #include <errno.h>
2570004 #include <stdint.h>
2570005 #include <arpa/inet.h>
2570006 //-------------------------------------------------------
2570007 int
2570008 net_index (h_addr_t ip)
2570009 {
2570010   //
2570011   int n;
2570012   //
2570013   // By IPv4 address.
2570014   //
2570015   if (ip != 0)
2570016     {
2570017       for (n = 0; n < NET_MAX_DEVICES; n++)
2570018         {
2570019           if (net_table[n].ip == ip)
2570020             {
2570021               return (n);
2570022             }
2570023         }
2570024     }
2570025   //
2570026   // Not found!
2570027   //
2570028   errset (ENODEV);
2570029   return (-1);
2570030 }
```

94.12.28 kernel/net/net_index_eth.c

Si veda la sezione 93.17.

```
2580001 #include <sys/os32.h>
2580002 #include <kernel/net.h>
2580003 #include <errno.h>
2580004 #include <stdint.h>
2580005 #include <arpa/inet.h>
2580006 //-------------------------------------------------------
2580007 int
2580008 net_index_eth (h_addr_t ip, uint8_t mac[6], uintptr_t io)
2580009 {
2580010   //
2580011   int n;
2580012   //
2580013   // If 'ip' is not zero, then find the Ethernet table
2580014   // index by that
2580015   // value.
2580016   //
2580017   if (ip != 0)
2580018     {
2580019       for (n = 0; n < NET_MAX_DEVICES; n++)
2580020         {
2580021           if (net_table[n].type & NET_DEV_ETH)
2580022             {
2580023               if (net_table[n].ip == ip)
2580024                 {
2580025                   return (n);
2580026                 }
2580027             }
2580028         }
2580029     }
2580030   //
2580031   // By mac address.
2580032   //
2580033   if (mac != NULL)
2580034     {
2580035       for (n = 0; n < NET_MAX_DEVICES; n++)
2580036         {
2580037           if (net_table[n].type & NET_DEV_ETH)
2580038             {
2580039               if (net_table[n].ethernet.mac[0] ==
2580040                   mac[0]
2580041                   && net_table[n].ethernet.mac[1] ==
2580042                   mac[1]
2580043                   && net_table[n].ethernet.mac[2] ==
2580044                   mac[2]
2580045                   && net_table[n].ethernet.mac[3] ==
2580046                   mac[3]
2580047                   && net_table[n].ethernet.mac[4] ==
2580048                   mac[4]
2580049                   && net_table[n].ethernet.mac[5] == mac[5])
2580050                 {
2580051                   return (n);
2580052                 }
2580053             }
2580054         }
2580055     }
2580056   //
2580057   // By hardware I/O address.
2580058   //
2580059   if (io > 0)
2580060     {
2580061       for (n = 0; n < NET_MAX_DEVICES; n++)
2580062         {
2580063           if (net_table[n].type & NET_DEV_ETH)
2580064             {
2580065               if (net_table[n].ethernet.base_io == io)
2580066                 {
2580067                   return (n);
2580068                 }
2580069             }
2580070         }
2580071     }
2580072   //
2580073   // Not found!
2580074   //
2580075   errset (ENODEV);
2580076   return (-1);
2580077 }
```

94.12.29 kernel/net/net_init.c

Si veda la sezione 93.17.

```
2590001 #include <kernel/net.h>
2590002 #include <kernel/net/route.h>
2590003 #include <kernel/net/arp.h>
2590004 #include <kernel/lib_s.h>
2590005 #include <kernel/proc.h>
2590006 #include <kernel/multiboot.h>
2590007 #include <stdlib.h>
2590008 #include <kernel/lib_k.h>
2590009 #include <string.h>
2590010 #include <errno.h>
2590011 #include <kernel/driver/pci.h>
2590012 #include <kernel/driver/nic/ne2k.h>
2590013 //-------------------------------------------------------
2590014 static void net_eth_init (int start);
```

```
2590015 //-------------------------------------------------------
2590016 void
2590017 net_init (void)
2590018 {
2590019   int n;        // NET device table index.
2590020   int b;        // Buffer NET device index.
2590021   int i;
2590022   char *net = "net0";
2590023   char *route = "route0";
2590024   char **argument;
2590025   in_addr_t ip_a;
2590026   in_addr_t ip_b;
2590027   int status;
2590028   //
2590029   // Reset the NET device table.
2590030   //
2590031   for (n = 0; n < NET_MAX_DEVICES; n++)
2590032     {
2590033       net_table[n].type = NET_DEV_NULL;
2590034     }
2590035   //
2590036   // Set up the loopback interface.
2590037   //
2590038   net_table[0].type = NET_DEV_LOOPBACK;
2590039   net_table[0].ip = INADDR_LOOPBACK;
2590040   net_table[0].m = 8;
2590041   //
2590042   for (b = 0; b < NET_MAX_BUFFERS; b++)
2590043     {
2590044       net_table[0].loopback.buffer[b].clock = 0;
2590045     }
2590046   //
2590047   // Prepare ARP table
2590048   //
2590049   arp_init ();
2590050   //
2590051   // Add Ethernet devices, but starting from the
2590052   // second interface
2590053   // inside the net_table[].
2590054   //
2590055   net_eth_init (1);
2590056   //
2590057   // Prepare routes.
2590058   //
2590059   route_init ();
2590060   route_sort ();
2590061   //
2590062   // Command line options: counter 'i' is scanned like
2590063   // a character.
2590064   //
2590065   for (i = '0'; i <= '9'; i++)
2590066     {
2590067       net[3] = i;
2590068       argument = mboot_cmdline_opt (net, ",");
2590069       if (argument != NULL)
2590070         {
2590071           //
2590072           status = inet_pton (AF_INET, argument[2], &ip_a);
2590073           if (status != 1)
2590074             {
2590075               continue;
2590076             }
2590077           //
2590078           s_ipconfig ((pid_t) 0, atoi (argument[1]),
2590079                       ip_a, atoi (argument[3]));
2590080         }
2590081     }
2590082   //
2590083   for (i = '0'; i <= '9'; i++)
2590084     {
2590085       route[5] = i;
2590086       argument = mboot_cmdline_opt (route, ",");
2590087       if (argument != NULL)
2590088         {
2590089           //
2590090           status = inet_pton (AF_INET, argument[1], &ip_a);
2590091           if (status != 1)
2590092             {
2590093               continue;
2590094             }
2590095           status = inet_pton (AF_INET, argument[3], &ip_b);
2590096           if (status != 1)
2590097             {
2590098               continue;
2590099             }
2590100           //
2590101           s_routeadd ((pid_t) 0,
2590102                       ip_a, atoi (argument[2]),
2590103                       ip_b, atoi (argument[4]));
2590104         }
2590105     }
2590106   //
2590107   //
2590108   //
2590109   net_print ();
2590110   route_print ();
2590111 }
2590112
2590113 //-------------------------------------------------------
2590114 static void
2590115 net_eth_init (int start)
2590116 {
2590117   int p;        // PCI table index.
2590118   int n;        // NET devices table index.
2590119   int i;
2590120   int j;
2590121   //
2590122   static const struct
2590123   {
2590124     unsigned short vendor;
2590125     unsigned short device;
2590126   } type_ne2k[] =
2590127   {
2590128     {
2590129     0x10ec, 0x8029},    // RealTek_RTL_8029
2590130     {
2590131     0x1050, 0x0940},    // Winbond_89C940
2590132     {
2590133     0x11f6, 0x1401},    // Compex_RL2000
2590134     {
2590135     0x8e2e, 0x3000},    // KTI_ET32P2
2590136     {
2590137     0x4a14, 0x5000},    // NetVin_NV5000SC
2590138     {
2590139     0x1106, 0x0926},    // Via_86C926
2590140     {
2590141     0x10bd, 0x0e34},    // SureCom_NE34
2590142     {
2590143     0x1050, 0x5a5a},    // Winbond_W89C940F
2590144     {
2590145     0x12c3, 0x0058},    // Holtek_HT80232
2590146     {
2590147     0x12c3, 0x5598},    // Holtek_HT80229
2590148     {
2590149     0x8c4a, 0x1980},    // Winbond_89C940_8c4a
2590150   };
2590151   //
2590152   //
2590153   //
2590154   n = start;
2590155   //
2590156   // Scan the PCI table and find NE2K Ethernet
2590157   // devices.
2590158   //
2590159   for (p = 0;
2590160        p < PCI_MAX_DEVICES && n < NET_MAX_DEVICES; p++)
2590161     {
2590162       for (i = 0; i < sizeof_array (type_ne2k); i++)
2590163         {
2590164           if (pci_table[p].vendor_id ==
2590165               type_ne2k[i].vendor
2590166               && pci_table[p].device_id ==
2590167               type_ne2k[i].device)
2590168             {
2590169               //
2590170               // Verify if the NIC is really a NE2K.
2590171               //
2590172               if (ne2k_check (pci_table[p].base_io) == 0)
2590173                 {
2590174                   //
2590175                   // Reset the NIC and get the
2590176                   // physical address.
2590177                   //
2590178                   if (ne2k_reset (pci_table[p].base_io,
2590179                                   net_table[n].
2590180                                   ethernet.mac) == 0)
2590181                     {
2590182                       //
2590183                       // New.
2590184                       //
2590185                       net_table[n].type = NET_DEV_ETH_NE2K;
2590186                       net_table[n].ethernet.base_io =
2590187                         pci_table[p].base_io;
2590188                       net_table[n].ethernet.irq =
```

```
2590189                         pci_table[p].irq;
2590190                       //
2590191                       for (j = 0; j < NET_MAX_BUFFERS; j++)
2590192                         {
2590193                           net_table[n].ethernet.
2590194                             buffer[j].clock = 0;
2590195                         }
2590196                       //
2590197                       // Go to next NET table element.
2590198                       //
2590199                       n++;
2590200                       //
2590201                       break;
2590202                     }
2590203                 }
2590204             }
2590205         }
2590206     }
2590207 }
```

94.12.30 kernel/net/net_print.c

Si veda la sezione 93.17.

```
2600001 #include <sys/os32.h>
2600002 #include <kernel/net.h>
2600003 #include <errno.h>
2600004 //-------------------------------------------------------
2600005 void
2600006 net_print (void)
2600007 {
2600008   int n;        // NET devices table index.
2600009   char string[80];
2600010   //
2600011   //
2600012   //
2600013   k_printf ("dev   "
2600014             "address/mask        "
2600015             "mac                " "io      irq\n");
2600016   //
2600017   for (n = 0; n < NET_MAX_DEVICES; n++)
2600018     {
2600019       if (net_table[n].type != NET_DEV_NULL)
2600020         {
2600021           sprintf (string, "net%i    ", n);
2600022           string[6] = '\0';
2600023           k_printf ("%s", string);
2600024           //
2600025           sprintf (string, "%i.%i.%i.%i/%i "
2600026                    "               ",
2600027                    net_table[n].ip >> 24 & 0x000000FF,
2600028                    net_table[n].ip >> 16 & 0x000000FF,
2600029                    net_table[n].ip >> 8 & 0x000000FF,
2600030                    net_table[n].ip >> 0 & 0x000000FF,
2600031                    net_table[n].m);
2600032           string[20] = '\0';
2600033           k_printf ("%s", string);
2600034           //
2600035           if (net_table[n].type & NET_DEV_ETH)
2600036             {
2600037               k_printf
2600038                 ("%02x:%02x:%02x:%02x:%02x:%02x  "
2600039                  "0x%04x  %i",
2600040                  net_table[n].ethernet.mac[0],
2600041                  net_table[n].ethernet.mac[1],
2600042                  net_table[n].ethernet.mac[2],
2600043                  net_table[n].ethernet.mac[3],
2600044                  net_table[n].ethernet.mac[4],
2600045                  net_table[n].ethernet.mac[5],
2600046                  net_table[n].ethernet.base_io,
2600047                  net_table[n].ethernet.irq);
2600048             }
2600049           k_printf ("\n");
2600050         }
2600051     }
2600052 }
```

94.12.31 kernel/net/net_public.c

Si veda la sezione 93.17.

```
2610001 #include <kernel/net.h>
2610002 //-------------------------------------------------------
2610003 net_t net_table[NET_MAX_DEVICES];
2610004 //-------------------------------------------------------
```


94.12.32 kernel/net/net_rx.c

Si veda la sezione 93.17.

```
2620001 #include <arpa/inet.h>
2620002 #include <kernel/net.h>
2620003 #include <kernel/net/arp.h>
2620004 #include <sys/os32.h>
2620005 #include <kernel/lib_k.h>
2620006 //-------------------------------------------------------
2620007 #define DEBUG 0
2620008 //-------------------------------------------------------
2620009 int
2620010 net_rx (void)
2620011 {
2620012   int n;        // NET table index.
2620013   int b;        // Frame index.
2620014   net_ethernet_frame_t *frame;
2620015   int counter = 0;
2620016   //
2620017   // Scan NET table.
2620018   //
2620019   for (n = 0; n < NET_MAX_DEVICES; n++)
2620020     {
2620021       //
2620022       // Ethernet.
2620023       //
2620024       if (net_table[n].type & NET_DEV_ETH)
2620025         {
2620026           for (b = 0; b < NET_MAX_BUFFERS; b++)
2620027             {
2620028               if (net_table[n].ethernet.buffer[b].clock > 0)
2620029                 {
2620030                   frame = (net_ethernet_frame_t *)
2620031                     & net_table[n].ethernet.buffer[b].frame;
2620032                   //
2620033                   if (ntohs (frame->header.type) ==
2620034                       NET_PROT_ARP)
2620035                     {
2620036                       arp_rx (n, b);
2620037                       //
2620038                       // Remove packet from buffer.
2620039                       //
2620040                       net_table[n].ethernet.buffer[b].
2620041                         clock = 0;
2620042                       //
2620043                       // Increment the packet received
2620044                       // counter.
2620045                       //
2620046                       counter++;
2620047                     }
2620048                   else if (ntohs (frame->header.type)
2620049                            == NET_PROT_IP)
2620050                     {
2620051                       ip_rx (n, b);
2620052                       //
2620053                       // Remove packet from buffer.
2620054                       //
2620055                       net_table[n].ethernet.buffer[b].
2620056                         clock = 0;
2620057                       //
2620058                       // Increment the packet received
2620059                       // counter.
2620060                       //
2620061                       counter++;
2620062                     }
2620063                   else
2620064                     {
2620065                       //
2620066                       // Unknown frame type.
2620067                       //
2620068                       k_printf
2620069                         ("received an unknown frame "
2620070                          "type %04x\n",
2620071                          ntohs (frame->header.type));
2620072                       //
2620073                       // Remove packet from buffer.
2620074                       //
2620075                       net_table[n].ethernet.buffer[b].
2620076                         clock = 0;
2620077                       //
2620078                       // Increment the packet received
2620079                       // counter anyway.
2620080                       //
2620081                       counter++;
2620082                     }
2620083                 }
2620084             }
2620085         }
```

```
2620086         //
2620087         // Loopback
2620088         //
2620089         else if (net_table[n].type & NET_DEV_LOOP)
2620090           {
2620091             for (b = 0; b < NET_MAX_BUFFERS; b++)
2620092               {
2620093                 if (net_table[n].loopback.buffer[b].clock > 0)
2620094                   {
2620095                     ip_rx (n, b);
2620096                     //
2620097                     // Remove packet from buffer.
2620098                     //
2620099                     net_table[n].loopback.buffer[b].clock = 0;
2620100                     //
2620101                     // Increment the packet received
2620102                     // counter.
2620103                     //
2620104                     counter++;
2620105                     //
2620106                   }
2620107               }
2620108           }
2620109       }
2620110     //
2620111     // Remove ARP items that are too old.
2620112     //
2620113     arp_clean ();
2620114     //
2620115     //
2620116     //
2620117     return (counter);
2620118 }
```

94.12.33 kernel/net/route.h

«

Si veda la sezione 93.21.

```
2630001 #ifndef _KERNEL_NET_ROUTE_H
2630002 #define _KERNEL_NET_ROUTE_H    1
2630003 //-------------------------------------------------------------
2630004 #include <stdint.h>
2630005 #include <sys/types.h>
2630006 #include <kernel/net.h>
2630007 #include <netinet/in.h>
2630008 //-------------------------------------------------------------
2630009 #define ROUTE_MAX_ROUTES 16
2630010 //
2630011 // Route table element.
2630012 //
2630013 typedef struct
2630014 {
2630015   h_addr_t network;     // 32 bit, host byte order.
2630016   h_addr_t netmask;     // 32 bit, host byte order.
2630017   h_addr_t router;      // 32 bit, host byte order.
2630018   uint8_t m;    // Short netmask.
2630019   uint8_t interface;
2630020 } route_t;
2630021 //
2630022 // External routing table data.
2630023 //
2630024 extern route_t route_table[ROUTE_MAX_ROUTES];
2630025 //-------------------------------------------------------------
2630026 void route_init (void);
2630027 void route_sort (void);
2630028 void route_print (void);
2630029 h_addr_t route_remote_to_local (h_addr_t remote);
2630030 h_addr_t route_remote_to_router (h_addr_t remote);
2630031
2630032 //-------------------------------------------------------------
2630033 #endif
```

94.12.34 kernel/net/route/route_init.c

«

Si veda la sezione 93.21.

```
2640001 #include <arpa/inet.h>
2640002 #include <sys/os32.h>
2640003 #include <kernel/net/route.h>
2640004 #include <errno.h>
2640005 #include <netinet/in.h>
2640006 //-------------------------------------------------------------
2640007 void
2640008 route_init (void)
2640009 {
2640010   //
2640011   // Reset the table with 0xFF.
```



```
2640012   //
2640013   memset (route_table, 0xFF, sizeof (route_table));
2640014   //
2640015   // Put the loopback routing.
2640016   //
2640017   route_table[0].netmask = 0xFF000000;  // Little
2640018   // endian.
2640019   route_table[0].m = 8;
2640020   route_table[0].network =
2640021     INADDR_LOOPBACK & route_table[0].netmask;
2640022   route_table[0].router = 0;
2640023   route_table[0].interface = 0;
2640024 }
```

94.12.35 kernel/net/route/route_print.c

«

Si veda la sezione 93.21.

```
2650001 #include <arpa/inet.h>
2650002 #include <sys/os32.h>
2650003 #include <kernel/net/route.h>
2650004 #include <kernel/lib_k.h>
2650005 #include <errno.h>
2650006 //-------------------------------------------------------------
2650007 void
2650008 route_print (void)
2650009 {
2650010   int r;        // Routing table index.
2650011   char string[80];
2650012   //
2650013   k_printf ("Destination/mask   "
2650014             "Router          " "Interface\n");
2650015   //
2650016   for (r = 0; r < ROUTE_MAX_ROUTES; r++)
2650017     {
2650018       if (route_table[r].network == 0xFFFFFFFF)
2650019         {
2650020           //
2650021           // Empty item.
2650022           //
2650023           continue;
2650024         }
2650025       //
2650026       sprintf (string, "%i.%i.%i.%i/%i"
2650027                "                    ",
2650028                route_table[r].network >> 24 & 0x000000FF,
2650029                route_table[r].network >> 16 & 0x000000FF,
2650030                route_table[r].network >> 8 & 0x000000FF,
2650031                route_table[r].network >> 0 & 0x000000FF,
2650032                route_table[r].m);
2650033       string[19] = '\0';
2650034       k_printf ("%s", string);
2650035       //
2650036       if (route_table[r].router == 0)
2650037         {
2650038           k_printf ("               ");
2650039         }
2650040       else
2650041         {
2650042           sprintf (string, "%i.%i.%i.%i"
2650043                    "          ",
2650044                    route_table[r].router >> 24 & 0x000000FF,
2650045                    route_table[r].router >> 16 & 0x000000FF,
2650046                    route_table[r].router >> 8 & 0x000000FF,
2650047                    route_table[r].router >> 0 & 0x000000FF);
2650048           string[16] = '\0';
2650049           k_printf ("%s", string);
2650050         }
2650051       //
2650052       k_printf ("net%i\n", route_table[r].interface);
2650053     }
2650054 }
```

94.12.36 kernel/net/route/route_public.c

«

Si veda la sezione 93.21.

```
2660001 #include <kernel/net/route.h>
2660002 //-------------------------------------------------------------
2660003 route_t route_table[ROUTE_MAX_ROUTES];
2660004 //-------------------------------------------------------------
```

94.12.37 kernel/net/route/route_remote_to_local.c

Si veda la sezione 93.21.

```
2670001 #include <arpa/inet.h>
2670002 #include <sys/os32.h>
2670003 #include <kernel/net/route.h>
2670004 #include <kernel/lib_k.h>
2670005 #include <errno.h>
2670006 //-------------------------------------------------------
2670007 h_addr_t
2670008 route_remote_to_local (h_addr_t remote)
2670009 {
2670010   int r;        // Routing table index.
2670011   int d;        // Network interface number.
2670012   h_addr_t network;
2670013   //
2670014   for (r = 0; r < ROUTE_MAX_ROUTES; r++)
2670015     {
2670016       //
2670017       // Calculate the remote network address based on
2670018       // the current
2670019       // router item netmask.
2670020       //
2670021       network = remote & route_table[r].netmask;
2670022       //
2670023       // Compare the calculated network address with
2670024       // the remote
2670025       // network.
2670026       //
2670027       if (route_table[r].network == network)
2670028         {
2670029           //
2670030           // Found.
2670031           //
2670032           d = route_table[r].interface;
2670033           //
2670034           // Check inside the network interfaces.
2670035           //
2670036           if (net_table[d].ip == 0)
2670037             {
2670038               errset (ENODEV);
2670039               return ((h_addr_t) - 1);
2670040             }
2670041           else
2670042             {
2670043               return (net_table[d].ip);
2670044             }
2670045         }
2670046     }
2670047   //
2670048   // Sorry: destination not found.
2670049   //
2670050   errset (EADDRNOTAVAIL);
2670051   return ((h_addr_t) - 1);
2670052 }
```

94.12.38 kernel/net/route/route_remote_to_router.c

Si veda la sezione 93.21.

```
2680001 #include <arpa/inet.h>
2680002 #include <sys/os32.h>
2680003 #include <kernel/net/route.h>
2680004 #include <kernel/lib_k.h>
2680005 #include <errno.h>
2680006 //-------------------------------------------------------
2680007 h_addr_t
2680008 route_remote_to_router (h_addr_t remote)
2680009 {
2680010   int r;        // Routing table index.
2680011   h_addr_t network;
2680012   //
2680013   for (r = 0; r < ROUTE_MAX_ROUTES; r++)
2680014     {
2680015       //
2680016       // Calculate the remote network address based on
2680017       // the current
2680018       // router item netmask.
2680019       //
2680020       network = remote & route_table[r].netmask;
2680021       //
2680022       // Compare the calculated network address with
2680023       // the remote
2680024       // network: routes are sorted, from the most
2680025       // detailed to the
2680026       // less one.
2680027       //
2680028       if (route_table[r].network == network)
2680029         {
2680030           //
2680031           // Found.
2680032           //
2680033           return (route_table[r].router);
2680034         }
2680035     }
2680036   //
2680037   // Sorry: destination not found.
2680038   //
2680039   errset (EADDRNOTAVAIL);
2680040   return ((h_addr_t) - 1);
2680041 }
```


94.12.39 kernel/net/route/route_sort.c

Si veda la sezione 93.21.

```
2690001 #include <sys/os32.h>
2690002 #include <kernel/net/route.h>
2690003 #include <errno.h>
2690004 //-------------------------------------------------------
2690005 static int part (char *array, size_t size, int a,
2690006                  int z, int (*compare) (void *, void *));
2690007 static void sort (char *array, size_t size, int a,
2690008                   int z, int (*compare) (void *, void *));
2690009 static void qsort (void *base, size_t nmemb,
2690010                    size_t size, int (*compare) (void *,
2690011                                                 void *));
2690012 //
2690013 static uint8_t swap[sizeof (route_t)];
2690014 static int comp (void *a, void *b);
2690015 //-------------------------------------------------------
2690016 void
2690017 route_sort (void)
2690018 {
2690019   qsort (route_table, ROUTE_MAX_ROUTES,
2690020          sizeof (route_t), comp);
2690021 }
2690022
2690023 //-------------------------------------------------------
2690024 static int
2690025 comp (void *a, void *b)
2690026 {
2690027   route_t *route_a = a;
2690028   route_t *route_b = b;
2690029   uint8_t m_a = route_a->m;
2690030   uint8_t m_b = route_b->m;
2690031   //
2690032   if (m_a > m_b)
2690033     return (-1);
2690034   if (m_a < m_b)
2690035     return (1);
2690036   return (0);
2690037 }
2690038
2690039 //-------------------------------------------------------
2690040 static void
2690041 qsort (void *base, size_t nmemb, size_t size,
2690042        int (*compare) (void *, void *))
2690043 {
2690044   if (size <= 1)
2690045     {
2690046       //
2690047       // There is nothing to sort!
2690048       //
2690049       return;
2690050     }
2690051   else
2690052     {
2690053       sort ((char *) base, size, 0, (int) (nmemb - 1),
2690054             compare);
2690055     }
2690056 }
2690057
2690058 //-------------------------------------------------------
2690059 static void
2690060 sort (char *array, size_t size, int a, int z,
2690061       int (*compare) (void *, void *))
2690062 {
2690063   int loc;
2690064   //
2690065   if (z > a)
2690066     {
2690067       loc = part (array, size, a, z, compare);
2690068       if (loc >= 0)
```

```
        {
          sort (array, size, a, loc - 1, compare);
          sort (array, size, loc + 1, z, compare);
        }
    }
}

//-------------------------------------------------------
static int
part (char *array, size_t size, int a, int z,
      int (*compare) (void *, void *))
{
  int i;
  int loc;
  //
  if (z <= a)
    {
      errset (EUNKNOWN);        // Should never
      // happen.
      return (-1);
    }
  //
  // Index 'i' after the first element; index 'loc' at
  // the last
  // position.
  //
  i = a + 1;
  loc = z;
  //
  // Loop as long as index 'loc' is higher than index
  // 'i'.
  // When index 'loc' is less or equal to index 'i',
  // then, index 'loc' is the right position for the
  // first element of the current piece of array.
  //
  for (;;)
    {
      //
      // Index 'i' goes up...
      //
      for (; i < loc; i++)
        {
          if (compare
              (&array[i * size], &array[a * size]) > 0)
            {
              break;
            }
        }
      //
      // Index 'loc' gose down...
      //
      for (;; loc--)
        {
          if (compare
              (&array[loc * size], &array[a * size]) <= 0)
            {
              break;
            }
        }
      //
      // Swap elements related to index 'i' and 'loc'.
      //
      if (loc <= i)
        {
          //
          // The array is completely scanned.
          //
          break;
        }
      else
        {
          memcpy (swap, &array[loc * size], size);
          memcpy (&array[loc * size], &array[i * size],
                  size);
          memcpy (&array[i * size], swap, size);
        }
    }
  //
  // Swap the first element with the one related to
  // the
  // index 'loc'.
  //
  memcpy (swap, &array[loc * size], size);
  memcpy (&array[loc * size], &array[a * size], size);
  memcpy (&array[a * size], swap, size);
  //
  // Return the index 'loc'.
  //
  return (loc);
}
```


94.12.40 kernel/net/tcp.h

«

Si veda la sezione 93.23.

```
#ifndef _KERNEL_NET_TCP_H
#define _KERNEL_NET_TCP_H    1
//-------------------------------------------------------
#include <netinet/tcp.h>
#include <kernel/net.h>
//-------------------------------------------------------
#define TCP_HEADER_SIZE      20
#define TCP_MAX_PACKET_SIZE  NET_IP_MAX_DATA_SIZE
#define TCP_MAX_DATA_SIZE    \
    TCP_MAX_PACKET_SIZE-TCP_HEADER_SIZE
//
#define TCP_MAX_DELAY        (CLOCKS_PER_SEC*2)
//-------------------------------------------------------
//
// TCP packet, for transmission.
//
typedef struct
{
  struct tcphdr header;
  uint8_t data[TCP_MAX_DATA_SIZE];
} __attribute__ ((packed)) tcp_packet_t;
//
// TCP pseudo header for checksum calculation.
//
typedef struct
{
  in_addr_t saddr;
  in_addr_t daddr;
  uint8_t zero;
  uint8_t protocol;
  uint16_t length;
} __attribute__ ((packed)) tcp_pseudo_header_t;
//-------------------------------------------------------
#define TCP_FLAG_NULL  0
#define TCP_FLAG_ACK   1
#define TCP_FLAG_PSH   2
#define TCP_FLAG_RST   4
#define TCP_FLAG_SYN   8
#define TCP_FLAG_FIN  16
//-------------------------------------------------------
#define TCP_TRY_READ   1        // 2^0 Wake up reading
                                // TCP process.
#define TCP_TRY_WRITE  2        // 2^1 Wake up writing
                                // TCP process.
//-------------------------------------------------------
int tcp_tx_raw (h_port_t sport, h_port_t dport,
                uint32_t seq, uint32_t ack_seq,
                int flags,
                h_addr_t saddr, h_addr_t daddr,
                const void *buffer, size_t size);
int tcp_tx_sock (void *sock_item);
int tcp_tx_ack (void *sock_item);
int tcp_rx_ack (void *sock_item, void *packet);
int tcp (void);
int tcp_connect (void *sock_item);
int tcp_tx_rst (void *ip_packet);
void tcp_test (void);
void tcp_show (h_addr_t src, h_addr_t dst,
               const struct tcphdr *tcphdr);
int tcp_close (void *sock_item);
int tcp_rx_data (void *sock_item, void *packet);
int tcp_status (void *ip_packet);
//-------------------------------------------------------
#endif
```

94.12.41 kernel/net/tcp/tcp.c

«

Si veda la sezione 93.23.

```
#include <stdlib.h>
#include <string.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <kernel/net.h>
#include <kernel/net/tcp.h>
#include <kernel/fs.h>
#include <kernel/lib_s.h>
#include <kernel/lib_k.h>
#include <errno.h>
```

```
2710011 //-------------------------------------------------------
2710012 #define DEBUG 0
2710013 //-------------------------------------------------------
2710014 int
2710015 tcp (void)
2710016 {
2710017   int s;        // Socket table index.
2710018   int p;        // IP table index.
2710019   int q;        // Queue index.
2710020   int status;
2710021   struct tcphdr *tcp;
2710022   struct iphdr *ip;
2710023   int sfdn;     // New socket.
2710024   fd_t *sfd;
2710025   sock_t *sock;
2710026   struct sockaddr_in sa;
2710027   clock_t delay;
2710028   uint8_t *recv_data;
2710029   size_t recv_size;
2710030   int ret = 0;
2710031   unsigned int lseq;
2710032   //
2710033   // Scan local sockets.
2710034   //
2710035   for (s = 0; s < SOCK_MAX_SLOTS; s++)
2710036     {
2710037       if (!sock_table[s].active)
2710038         continue;
2710039       if (sock_table[s].family != AF_INET)
2710040         continue;
2710041       if (sock_table[s].protocol != IPPROTO_TCP)
2710042         continue;
2710043       if (sock_table[s].unreach_port)
2710044         continue;
2710045       if (sock_table[s].unreach_host)
2710046         continue;
2710047       //
2710048       // Calculate the delay from the last send.
2710049       //
2710050       delay = s_clock ((pid_t) 0) - sock_table[s].tcp.clock;
2710051       //
2710052       // Have we received something? Scan the
2710053       // ip_table[] to find a
2710054       // TCP packet that was not already seen by the
2710055       // socket.
2710056       //
2710057       for (p = 0; p < IP_MAX_PACKETS; p++)
2710058         {
2710059           // ////////////////////////////////////////
2710060           // PACKET CHECK
2710061           // ////////////////////////////////////////
2710062           //
2710063           // Check the protocol.
2710064           //
2710065           if (ip_table[p].packet.header.protocol !=
2710066               IPPROTO_TCP)
2710067             {
2710068               //
2710069               // It is not TCP.
2710070               //
2710071               continue;
2710072             }
2710073           //
2710074           // Is the packet new for the socket?
2710075           //
2710076           if (ip_table[p].clock <=
2710077               sock_table[s].read.clock[p])
2710078             {
2710079               //
2710080               // Already seen or packet too old.
2710081               //
2710082               continue;
2710083             }
2710084           //
2710085           // Get a pointer to IP and TCP headers.
2710086           //
2710087           ip = (struct iphdr *) &ip_table[p].packet.header;
2710088           tcp =
2710089             (struct tcphdr *) &ip_table[p].packet.
2710090             octet[ip->ihl * 4];
2710091           //
2710092           // Verify the ports.
2710093           //
2710094           if (tcp->dest != htons (sock_table[s].lport))
2710095             {
2710096               //
2710097               // The local port does not match!
2710098               //
2710099               continue;
2710100             }
2710101           //
2710102           if (tcp->source != htons (sock_table[s].rport))
2710103             {
2710104               //
2710105               // The remote port does not match, but
2710106               // might be
2710107               // listening and the packet might be a
2710108               // SYN.
2710109               //
2710110               if (sock_table[s].rport == 0
2710111                   && sock_table[s].tcp.conn ==
2710112                   TCP_LISTEN && tcp->syn && !tcp->ack)
2710113                 {
2710114                   //
2710115                   // We hope that it is the first SYN.
2710116                   //
2710117                   ;
2710118                 }
2710119               else
2710120                 {
2710121                   continue;
2710122                 }
2710123             }
2710124           //
2710125           // Verify the IP addresses.
2710126           //
2710127           if (ip_table[p].packet.header.daddr
2710128               != htonl (sock_table[s].laddr))
2710129             {
2710130               //
2710131               // The local address does not match, but
2710132               // might be zero.
2710133               //
2710134               if (sock_table[s].laddr != 0)
2710135                 {
2710136                   //
2710137                   // The local address is not zero, so
2710138                   // the match fails.
2710139                   //
2710140                   continue;
2710141                 }
2710142             }
2710143           //
2710144           if (ip_table[p].packet.header.saddr
2710145               != htonl (sock_table[s].raddr))
2710146             {
2710147               //
2710148               // The remote address does not match,
2710149               // but the socket
2710150               // might be listening and che packet
2710151               // might be a SYN.
2710152               //
2710153               if (sock_table[s].raddr == 0
2710154                   && sock_table[s].tcp.conn ==
2710155                   TCP_LISTEN && tcp->syn && !tcp->ack)
2710156                 {
2710157                   //
2710158                   // We hope that it is the first SYN.
2710159                   //
2710160                   ;
2710161                 }
2710162               else
2710163                 {
2710164                   continue;
2710165                 }
2710166             }
2710167           //
2710168           // This TCP packet is new for the socket:
2710169           // save the clock time, so that the
2710170           // same packet is not read again.
2710171           //
2710172           sock_table[s].read.clock[p] = ip_table[p].clock;
2710173           //
2710174           // ////////////////////////////////////////
2710175           // TCP PROTOCOL
2710176           // ////////////////////////////////////////
2710177           //
2710178           if (DEBUG)
2710179             {
2710180               tcp_show (ntohl (ip->saddr),
2710181                         ntohl (ip->daddr), tcp);
2710182             }
2710183           //
2710184           recv_data = &((uint8_t *) tcp)[tcp->doff * 4];
```

```
2710185             recv_size =
2710186               ntohs (ip->tot_len) - (ip->ihl * 4) -
2710187               (tcp->doff * 4);
2710188             //
2710189             // Now we have received a TCP packet for the
2710190             // current
2710191             // socket, and we should do something with
2710192             // it...
2710193             //
2710194             if (tcp->rst)
2710195               {
2710196                 //
2710197                 // We have received a reset... What is
2710198                 // resetting?
2710199                 //
2710200                 if ((sock_table[s].tcp.
2710201                      rsq[sock_table[s].tcp.rsqi] ==
2710202                      ntohl (tcp->seq)) || (tcp->ack
2710203                                            &&
2710204                                            (sock_table[s].tcp.
2710205                                             lsq_ack ==
2710206                                             ntohl (tcp->
2710207                                                    ack_seq))))
2710208                   {
2710209                     sock_table[s].tcp.recv_closed = 1;
2710210                     sock_table[s].tcp.send_closed = 1;
2710211                     sock_table[s].tcp.conn = TCP_RESET;
2710212                     if (DEBUG)
2710213                       {
2710214                         k_printf ("[%s] TCP_RESET\n",
2710215                                   __func__);
2710216                       }
2710217                   }
2710218                 else
2710219                   {
2710220                     k_printf ("lsq_ack=%3, rsq=%3\n",
2710221                               sock_table[s].tcp.lsq_ack,
2710222                               sock_table[s].tcp.
2710223                               rsq[sock_table[s].tcp.rsqi]);
2710224                   }
2710225               }
2710226             else if (sock_table[s].tcp.conn == 0
2710227                      || sock_table[s].tcp.conn ==
2710228                      TCP_CLOSE
2710229                      || sock_table[s].tcp.conn == TCP_RESET)
2710230               {
2710231                 //
2710232                 // The connection is not yet ready or it
2710233                 // is closed.
2710234                 // We are not waiting any packet, so we
2710235                 // just reject it.
2710236                 //
2710237                 tcp_tx_rst (ip);
2710238               }
2710239             else if (sock_table[s].tcp.conn == TCP_LISTEN)
2710240               {
2710241                 //
2710242                 // It should be a first SYN packet.
2710243                 //
2710244                 if (tcp->syn && !tcp->ack)
2710245                   {
2710246                     //
2710247                     // Should be a new connection
2710248                     // attempt. Can we queue
2710249                     // it?
2710250                     //
2710251                     for (q = 0;
2710252                          q <
2710253                          sock_table[s].tcp.listen_max; q++)
2710254                       {
2710255                         if (sock_table[s].tcp.
2710256                             listen_queue[q] == -1)
2710257                           {
2710258                             break;
2710259                           }
2710260                       }
2710261                     if (q >= sock_table[s].tcp.listen_max)
2710262                       {
2710263                         //
2710264                         // The queue is full.
2710265                         //
2710266                         tcp_tx_rst (ip);
2710267                         //
2710268                         // Next packet.
2710269                         //
2710270                         continue;
2710271                       }
```

```
2710272                     //
2710273                     // Is this connection attempt
2710274                     // already done?
2710275                     //
2710276                     status = tcp_status (ip);
2710277                     //
2710278                     if (status < 0)
2710279                       {
2710280                         //
2710281                         // Should not happen.
2710282                         //
2710283                         errset (errno);
2710284                         perror (NULL);
2710285                         //
2710286                         // Ignore the packet?!
2710287                         //
2710288                         continue;
2710289                       }
2710290                     else if (status == TCP_SYN_SENT)
2710291                       {
2710292                         //
2710293                         // The same SYN was already
2710294                         // received and serviced:
2710295                         // just ignore the packet.
2710296                         //
2710297                         continue;
2710298                       }
2710299                     else if (status > 0)
2710300                       {
2710301                         //
2710302                         // There is already a connection
2710303                         // with the same
2710304                         // addresses: ignore the SYN.
2710305                         //
2710306                         continue;
2710307                       }
2710308                     //
2710309                     // The SYN is new!
2710310                     // Can we open a new socket?
2710311                     //
2710312                     sfdn =
2710313                       s_socket (sock_table[s].tcp.listen_pid,
2710314                                 AF_INET, SOCK_STREAM,
2710315                                 IPPROTO_TCP);
2710316                     if (sfdn < 0)
2710317                       {
2710318                         //
2710319                         // No, sorry.
2710320                         //
2710321                         tcp_tx_rst (ip);
2710322                         //
2710323                         // Next packet.
2710324                         //
2710325                         continue;
2710326                       }
2710327                     //
2710328                     // Can we bind it to the same
2710329                     // destination of the
2710330                     // received packet?
2710331                     //
2710332                     sa.sin_family = AF_INET;
2710333                     sa.sin_port = tcp->dest;
2710334                     sa.sin_addr.s_addr = ip->daddr;
2710335                     status =
2710336                       s_bind (sock_table[s].tcp.listen_pid,
2710337                               sfdn, (struct sockaddr *) &sa,
2710338                               sizeof (sa));
2710339                     if (status < 0)
2710340                       {
2710341                         //
2710342                         // No, sorry.
2710343                         //
2710344                         tcp_tx_rst (ip);
2710345                         close (sfdn);
2710346                         //
2710347                         // Next packet.
2710348                         //
2710349                         continue;
2710350                       }
2710351                     //
2710352                     // Ok. Save the new socket number in
2710353                     // queue.
2710354                     //
2710355                     sock_table[s].tcp.listen_queue[q] = sfdn;
2710356                     //
2710357                     // Prepare some pointers to reach
2710358                     // the new socket
```

```
2710359                     // easily.
2710360                     //
2710361                     sfd =
2710362                       fd_reference (sock_table[s].tcp.
2710363                                     listen_pid, &sfdn);
2710364                     sock = sfd->file->sock;
2710365                     //
2710366                     // Connect the new socket with the
2710367                     // remote node.
2710368                     //
2710369                     sock->raddr = ntohl (ip->saddr);
2710370                     sock->rport = ntohs (tcp->source);
2710371                     //
2710372                     // The new socket has seen this SYN
2710373                     // packet.
2710374                     //
2710375                     sock->read.clock[p] = ip_table[p].clock;
2710376                     //
2710377                     // Make the new socket answare with
2710378                     // a second
2710379                     // SYN.
2710380                     //
2710381                     memset (sock->tcp.lsq, 0x00,
2710382                             sizeof (sock->tcp.lsq));
2710383                     memset (sock->tcp.rsq, 0x00,
2710384                             sizeof (sock->tcp.rsq));
2710385                     srand ((unsigned int)
2710386                            s_clock ((pid_t) 0));
2710387                     lseq = rand ();
2710388                     sock->tcp.lsq_ack = lseq + 1;
2710389                     sock->tcp.lsq[++sock->tcp.lsqi] = lseq;
2710390                     sock->tcp.rsq[++sock->tcp.rsqi] =
2710391                       ntohl (tcp->seq);
2710392                     sock->tcp.rsq[++sock->tcp.rsqi] =
2710393                       ntohl (tcp->seq) + 1;
2710394                     //
2710395                     sock->tcp.can_send = 1;
2710396                     sock->tcp.send_flags =
2710397                       TCP_FLAG_SYN | TCP_FLAG_ACK;
2710398                     if (DEBUG)
2710399                       {
2710400                         k_printf
2710401                           ("[%s] New conn. seq=%3u, "
2710402                            "lsq_ack=%3u\n",
2710403                            __func__, lseq, sock->tcp.lsq_ack);
2710404                       }
2710405                     tcp_tx_sock (sock);
2710406                     //
2710407                     // Put the new socket to
2710408                     // TCP_SYN_RECV status.
2710409                     //
2710410                     sock->tcp.conn = TCP_SYN_RECV;
2710411                   }
2710412                 else
2710413                   {
2710414                     //
2710415                     // We are listening: cannot accept
2710416                     // other type of
2710417                     // packets.
2710418                     //
2710419                     tcp_tx_rst (ip);
2710420                   }
2710421               }
2710422             else if (sock_table[s].tcp.conn == TCP_SYN_SENT)
2710423               {
2710424                 //
2710425                 // It should be a second SYN packet with
2710426                 // ACK.
2710427                 //
2710428                 if (tcp->syn
2710429                     && tcp->ack
2710430                     && tcp_rx_ack (&sock_table[s], ip) == 0)
2710431                   {
2710432                     //
2710433                     // SYN + ACK.
2710434                     //
2710435                     // Save the initial remote sequence,
2710436                     // because it
2710437                     // is the first one, and save also
2710438                     // the remote
2710439                     // sequence that we will expect next
2710440                     // time.
2710441                     //
2710442                     sock_table[s].tcp.rsq[++sock_table[s].
2710443                                           tcp.rsqi] =
2710444                       ntohl (tcp->seq);
2710445                     sock_table[s].tcp.rsq[++sock_table[s].
2710446                                           tcp.rsqi] =
2710447                       ntohl (tcp->seq) + 1;
2710448                     //
2710449                     // The received SYN is to be
2710450                     // confirmed with ACK.
2710451                     // The expected next local sequence
2710452                     // does not change,
2710453                     // and in effect, we don't expect
2710454                     // any other ACK back.
2710455                     //
2710456                     sock_table[s].tcp.lsq[++sock_table[s].
2710457                                           tcp.lsqi] =
2710458                       sock_table[s].tcp.lsq_ack;
2710459                     tcp_tx_ack (&sock_table[s]);
2710460                     //
2710461                     // We are now in TCP_ESTABLISHED.
2710462                     //
2710463                     sock_table[s].tcp.conn = TCP_ESTABLISHED;
2710464                     //
2710465                     // Now the process can write and can
2710466                     // receive
2710467                     // data (can write --but cannot
2710468                     // send-- and can
2710469                     // receive --but cannot read--).
2710470                     //
2710471                     sock_table[s].tcp.can_write = 1;
2710472                     sock_table[s].tcp.can_send = 0;
2710473                     //
2710474                     sock_table[s].tcp.can_recv = 1;
2710475                     sock_table[s].tcp.can_read = 0;
2710476                     //
2710477                     ret |= TCP_TRY_WRITE;
2710478                   }
2710479                 //
2710480                 // The case of a single ACK and a single
2710481                 // SYN is not
2710482                 // taken into consideration!
2710483                 //
2710484                 // No other type of packet is expected
2710485                 // here.
2710486                 //
2710487               }
2710488             else if (sock_table[s].tcp.conn == TCP_SYN_RECV)
2710489               {
2710490                 //
2710491                 // We are waiting an ACK for our second
2710492                 // SYN.
2710493                 //
2710494                 if (tcp->ack
2710495                     && tcp_rx_ack (&sock_table[s], ip) == 0)
2710496                   {
2710497                     //
2710498                     // ACK ok.
2710499                     // The connection is ready.
2710500                     //
2710501                     sock_table[s].tcp.conn = TCP_ESTABLISHED;
2710502                     //
2710503                     // Now the process can write and can
2710504                     // receive
2710505                     // data (can write --but cannot
2710506                     // send-- and can
2710507                     // receive --but cannot read--).
2710508                     //
2710509                     sock_table[s].tcp.can_write = 1;
2710510                     sock_table[s].tcp.can_send = 0;
2710511                     //
2710512                     sock_table[s].tcp.can_recv = 1;
2710513                     sock_table[s].tcp.can_read = 0;
2710514                     //
2710515                     ret |= TCP_TRY_WRITE;
2710516                   }
2710517                 //
2710518                 // No other type of packet is expected
2710519                 // here.
2710520                 //
2710521               }
2710522             else if (sock_table[s].tcp.conn ==
2710523                      TCP_ESTABLISHED)
2710524               {
2710525                 //
2710526                 // Might be a repeated SYN + ACK,
2710527                 // because the other
2710528                 // side don't have received our ACK.
2710529                 // Just resend.
2710530                 //
2710531                 if (tcp->syn && tcp->ack)
2710532                   {
```

```
2710533                     tcp_tx_ack (&sock_table[s]);
2710534                     //
2710535                     // Next packet.
2710536                     //
2710537                     continue;
2710538                   }
2710539                 //
2710540                 // It might be a normal ACK.
2710541                 //
2710542                 if (tcp->ack)
2710543                   {
2710544                     //
2710545                     // Verify if the packet contains
2710546                     // data: if there is
2710547                     // data, before sending the ACK,
2710548                     // must verify to be
2710549                     // able to receive such data.
2710550                     //
2710551                     if (recv_size > 0)
2710552                       {
2710553                         // k_printf ("[%i]", (int)
2710554                         // recv_size);
2710555                         //
2710556                         // There is data.
2710557                         //
2710558                         if (!sock_table[s].tcp.can_recv)
2710559                           {
2710560                             //
2710561                             // At the moment, cannot
2710562                             // receive: the packet
2710563                             // is currently ignored and
2710564                             // no ACK is sent.
2710565                             //
2710566                             continue;
2710567                           }
2710568                       }
2710569                     //
2710570                     // The received packet is empty or
2710571                     // it can be received.
2710572                     //
2710573                     if (tcp_rx_ack (&sock_table[s], ip) == 0)
2710574                       {
2710575                         //
2710576                         // ACK ok.
2710577                         // The process can continue to
2710578                         // write and the packet
2710579                         // don't have to be resent.
2710580                         //
2710581                         sock_table[s].tcp.can_write = 1;
2710582                         sock_table[s].tcp.can_send = 0;
2710583                         //
2710584                         ret |= TCP_TRY_WRITE;
2710585                       }
2710586                     else
2710587                       {
2710588                         //
2710589                         // Next packet.
2710590                         //
2710591                         continue;
2710592                       }
2710593                   }
2710594                 //
2710595                 // It might be a FIN.
2710596                 //
2710597                 if (tcp->fin)
2710598                   {
2710599                     //
2710600                     // Is the FIN in the right sequence?
2710601                     //
2710602                     if (sock_table[s].tcp.
2710603                         rsq[sock_table[s].tcp.rsqi] ==
2710604                         ntohl (tcp->seq))
2710605                       {
2710606                         //
2710607                         // Yes, it is: close receiving.
2710608                         //
2710609                         sock_table[s].tcp.recv_closed = 1;
2710610                         //
2710611                         // ACK.
2710612                         //
2710613                         sock_table[s].tcp.
2710614                           rsq[++sock_table[s].tcp.rsqi] =
2710615                           ntohl (tcp->seq) + 1;
2710616                         sock_table[s].tcp.
2710617                           lsq[++sock_table[s].tcp.lsqi] =
2710618                           sock_table[s].tcp.lsq_ack;
2710619                         tcp_tx_ack (&sock_table[s]);
```

```
2710620                         //
2710621                         // Change status.
2710622                         //
2710623                         if (DEBUG)
2710624                           {
2710625                             k_printf
2710626                               ("[%s] TCP_CLOSE_WAIT\n",
2710627                                __func__);
2710628                           }
2710629                         sock_table[s].tcp.conn =
2710630                           TCP_CLOSE_WAIT;
2710631                       }
2710632                     else
2710633                       {
2710634                         //
2710635                         // Just ignore it and jump to
2710636                         // the next packet.
2710637                         //
2710638                         continue;
2710639                       }
2710640                   }
2710641                 //
2710642                 // The received packet might contain
2710643                 // some data, but can
2710644                 // accept data only if the receiving
2710645                 // buffer is empty
2710646                 // (was already read from the reading
2710647                 // process).
2710648                 //
2710649                 tcp_rx_data (&sock_table[s], ip);
2710650                 //
2710651                 ret |= TCP_TRY_READ;
2710652               }
2710653             else if (sock_table[s].tcp.conn == TCP_CLOSE_WAIT)
2710654               {
2710655                 //
2710656                 // It might be an ACK.
2710657                 //
2710658                 if (tcp->ack)
2710659                   {
2710660                     if (tcp_rx_ack (&sock_table[s], ip) == 0)
2710661                       {
2710662                         //
2710663                         // ACK ok.
2710664                         // The process can continue to
2710665                         // write and the packet
2710666                         // don't have to be resent.
2710667                         //
2710668                         sock_table[s].tcp.can_write = 1;
2710669                         sock_table[s].tcp.can_send = 0;
2710670                         //
2710671                         ret |= TCP_TRY_WRITE;
2710672                       }
2710673                     else
2710674                       {
2710675                         //
2710676                         // Next packet.
2710677                         //
2710678                         continue;
2710679                       }
2710680                   }
2710681                 //
2710682                 // The data coming from the outside is
2710683                 // not taken anymore.
2710684                 //
2710685               }
2710686             else if (sock_table[s].tcp.conn == TCP_LAST_ACK)
2710687               {
2710688                 //
2710689                 // It might be the final ACK.
2710690                 //
2710691                 if (tcp->ack)
2710692                   {
2710693                     if (tcp_rx_ack (&sock_table[s], ip) == 0)
2710694                       {
2710695                         //
2710696                         // ACK ok. The two directions
2710697                         // are closed and
2710698                         // the packet confirmed don't
2710699                         // have to be resent.
2710700                         //
2710701                         sock_table[s].tcp.recv_closed = 1;
2710702                         sock_table[s].tcp.send_closed = 1;
2710703                         sock_table[s].tcp.can_send = 0;
2710704                         //
2710705                         // Change status.
2710706                         //
```

```
2710707                     if (DEBUG)
2710708                       {
2710709                         k_printf ("[%s] TCP_CLOSE\n",
2710710                                   __func__);
2710711                       }
2710712                     sock_table[s].tcp.conn = TCP_CLOSE;
2710713                   }
2710714                 else
2710715                   {
2710716                     //
2710717                     // Next packet.
2710718                     //
2710719                     continue;
2710720                   }
2710721               }
2710722             //
2710723             // The data coming from the outside is
2710724             // not taken anymore.
2710725             //
2710726           }
2710727         else if (sock_table[s].tcp.conn == TCP_FIN_WAIT1)
2710728           {
2710729             if (tcp->ack && tcp->fin)
2710730               {
2710731                 //
2710732                 // ACK and FIN
2710733                 //
2710734                 if (tcp_rx_ack (&sock_table[s], ip) == 0)
2710735                   {
2710736                     //
2710737                     // ACK ok: the confirmed packet
2710738                     // don't have to
2710739                     // be resent.
2710740                     //
2710741                     sock_table[s].tcp.conn =
2710742                       TCP_FIN_WAIT2;
2710743                     sock_table[s].tcp.can_send = 0;
2710744                   }
2710745                 else
2710746                   {
2710747                     //
2710748                     // Next packet.
2710749                     //
2710750                     continue;
2710751                   }
2710752                 //
2710753                 // Is the FIN in the right sequence?
2710754                 //
2710755                 if (sock_table[s].tcp.
2710756                     rsq[sock_table[s].tcp.rsqi] ==
2710757                     ntohl (tcp->seq))
2710758                   {
2710759                     //
2710760                     // Yes, it is: close receiving.
2710761                     //
2710762                     sock_table[s].tcp.recv_closed = 1;
2710763                     //
2710764                     // ACK.
2710765                     //
2710766                     sock_table[s].tcp.
2710767                       rsq[++sock_table[s].tcp.rsqi] =
2710768                       ntohl (tcp->seq) + 1;
2710769                     sock_table[s].tcp.
2710770                       lsq[++sock_table[s].tcp.lsqi] =
2710771                       sock_table[s].tcp.lsq_ack;
2710772                     tcp_tx_ack (&sock_table[s]);
2710773                     //
2710774                     // Change status.
2710775                     //
2710776                     if (DEBUG)
2710777                       {
2710778                         k_printf
2710779                           ("[%s] TCP_TIME_WAIT\n",
2710780                            __func__);
2710781                       }
2710782                     sock_table[s].tcp.conn =
2710783                       TCP_TIME_WAIT;
2710784                   }
2710785                 else
2710786                   {
2710787                     //
2710788                     // Just ignore it and jump to
2710789                     // the next packet.
2710790                     //
2710791                     continue;
2710792                   }
2710793               }
```

```
2710794               else if (tcp->ack)
2710795                 {
2710796                   //
2710797                   // ACK only.
2710798                   //
2710799                   if (tcp_rx_ack (&sock_table[s], ip) == 0)
2710800                     {
2710801                       //
2710802                       // ACK ok: the confirmed packet
2710803                       // don't have to
2710804                       // be resent.
2710805                       //
2710806                       sock_table[s].tcp.conn =
2710807                         TCP_FIN_WAIT2;
2710808                       sock_table[s].tcp.can_send = 0;
2710809                     }
2710810                   else
2710811                     {
2710812                       //
2710813                       // Next packet.
2710814                       //
2710815                       continue;
2710816                     }
2710817                 }
2710818               //
2710819               // The received packet might contain
2710820               // some data, but can
2710821               // accept data only if the receive
2710822               // channel is open and
2710823               // if the receiving buffer is empty.
2710824               //
2710825               tcp_rx_data (&sock_table[s], ip);
2710826               //
2710827               ret |= TCP_TRY_READ;
2710828             }
2710829           else if (sock_table[s].tcp.conn == TCP_FIN_WAIT2)
2710830             {
2710831               //
2710832               // It might be the final ACK.
2710833               //
2710834               if (tcp->fin && tcp->ack)
2710835                 {
2710836                   if (tcp_rx_ack (&sock_table[s], ip) == 0)
2710837                     {
2710838                       //
2710839                       // ACK ok: the packet don't have
2710840                       // to be resent.
2710841                       //
2710842                       sock_table[s].tcp.conn =
2710843                         TCP_TIME_WAIT;
2710844                       sock_table[s].tcp.can_send = 0;
2710845                       //
2710846                       // Next packet.
2710847                       //
2710848                       continue;
2710849                     }
2710850                   else
2710851                     {
2710852                       //
2710853                       // Next packet.
2710854                       //
2710855                       continue;
2710856                     }
2710857                 }
2710858               //
2710859               // The received packet might contain
2710860               // some data, but can
2710861               // accept data only if the receiving
2710862               // buffer is empty
2710863               // (was already read from the reading
2710864               // process).
2710865               //
2710866               tcp_rx_data (&sock_table[s], ip);
2710867               //
2710868               ret |= TCP_TRY_READ;
2710869             }
2710870           else if (sock_table[s].tcp.conn == TCP_TIME_WAIT)
2710871             {
2710872               //
2710873               // It might be duplicate final ACK.
2710874               //
2710875               if (tcp->fin && tcp->ack)
2710876                 {
2710877                   //
2710878                   // Just resend the ACK.
2710879                   //
2710880                   tcp_tx_ack (&sock_table[s]);
```

```
2710881                 }
2710882               //
2710883               // Close receiving too, if it is not
2710884               // already done.
2710885               //
2710886               sock_table[s].tcp.recv_closed = 1;
2710887               //
2710888               // Change status, without waiting
2710889               // anymore.
2710890               //
2710891               if (DEBUG)
2710892                 {
2710893                   k_printf ("[%s] TCP_CLOSE\n", __func__);
2710894                 }
2710895               sock_table[s].tcp.conn = TCP_CLOSE;
2710896             }
2710897         }
2710898       //
2710899       // See if there is something to be re-sent (if
2710900       // the flag
2710901       // 'tcp.can_send' is set).
2710902       //
2710903       if (sock_table[s].tcp.can_send
2710904           && delay > TCP_MAX_DELAY)
2710905         {
2710906           tcp_tx_sock (&sock_table[s]);
2710907         }
2710908     }
2710909   //
2710910   // Return.
2710911   //
2710912   return (ret);
2710913 }
```

94.12.42 kernel/net/tcp/tcp_close.c

Si veda la sezione 93.23.

```
2720001 #include <stdlib.h>
2720002 #include <netinet/ip.h>
2720003 #include <netinet/tcp.h>
2720004 #include <errno.h>
2720005 #include <kernel/net.h>
2720006 #include <kernel/net/tcp.h>
2720007 #include <kernel/fs.h>
2720008 #include <kernel/lib_k.h>
2720009 //-------------------------------------------------------
2720010 #define DEBUG 0
2720011 //-------------------------------------------------------
2720012 int
2720013 tcp_close (void *sock_item)
2720014 {
2720015   sock_t *sock = sock_item;
2720016   //
2720017   // Is there already a connection?
2720018   //
2720019   if (sock->tcp.conn == 0 || sock->tcp.conn == TCP_CLOSE)
2720020     {
2720021       //
2720022       // Done or never opened.
2720023       //
2720024       return (0);
2720025     }
2720026   else if (sock->tcp.conn == TCP_RESET)
2720027     {
2720028       //
2720029       // Change to closed.
2720030       //
2720031       if (DEBUG)
2720032         {
2720033           k_printf ("[%s] TCP_CLOSE\n", __func__);
2720034         }
2720035       sock->tcp.conn = TCP_CLOSE;
2720036       return (0);
2720037     }
2720038   else if (sock->tcp.conn == TCP_TIME_WAIT)
2720039     {
2720040       //
2720041       // Assume that the time is elapsed.
2720042       //
2720043       sock->tcp.conn = TCP_CLOSE;
2720044       if (DEBUG)
2720045         {
2720046           k_printf ("[%s] TCP_CLOSE\n", __func__);
2720047         }
2720048       sock->tcp.conn = TCP_CLOSE;
2720049       return (0);
```



```
2720050     }
2720051   else if (sock->tcp.conn == TCP_FIN_WAIT1)
2720052     {
2720053       errset (EALREADY);
2720054       return (-1);
2720055     }
2720056   else if (sock->tcp.conn == TCP_FIN_WAIT2)
2720057     {
2720058       errset (EALREADY);
2720059       return (-1);
2720060     }
2720061   else if (sock->tcp.conn == TCP_ESTABLISHED)
2720062     {
2720063       sock->tcp.can_send = 1;
2720064       sock->tcp.lsq[++sock->tcp.lsqi] = sock->tcp.lsq_ack;
2720065       sock->tcp.send_flags = TCP_FLAG_FIN | TCP_FLAG_ACK;
2720066       tcp_tx_sock (sock);
2720067       sock->tcp.conn = TCP_FIN_WAIT1;
2720068       if (DEBUG)
2720069         k_printf ("[%s] TCP_FIN_WAIT1\n", __func__);
2720070       errset (EINPROGRESS);
2720071       return (-1);
2720072     }
2720073   else if (sock->tcp.conn == TCP_CLOSE_WAIT)
2720074     {
2720075       sock->tcp.can_send = 1;
2720076       sock->tcp.lsq[++sock->tcp.lsqi] = sock->tcp.lsq_ack;
2720077       sock->tcp.send_flags = TCP_FLAG_FIN | TCP_FLAG_ACK;
2720078       tcp_tx_sock (sock);
2720079       sock->tcp.conn = TCP_LAST_ACK;
2720080       if (DEBUG)
2720081         k_printf ("[%s] TCP_LAST_ACK\n", __func__);
2720082       errset (EINPROGRESS);
2720083       return (-1);
2720084     }
2720085   else
2720086     {
2720087       sock->tcp.can_send = 1;
2720088       sock->tcp.lsq[++sock->tcp.lsqi] = sock->tcp.lsq_ack;
2720089       sock->tcp.send_flags = TCP_FLAG_FIN | TCP_FLAG_ACK;
2720090       tcp_tx_sock (sock);
2720091       sock->tcp.conn = TCP_CLOSE;
2720092       if (DEBUG)
2720093         k_printf ("[%s] TCP_CLOSE\n", __func__);
2720094       return (0);
2720095     }
2720096 }
```

94.12.43 kernel/net/tcp/tcp_connect.c

Si veda la sezione 93.23.

```
2730001 #include <stdlib.h>
2730002 #include <netinet/ip.h>
2730003 #include <netinet/tcp.h>
2730004 #include <errno.h>
2730005 #include <kernel/net.h>
2730006 #include <kernel/net/tcp.h>
2730007 #include <kernel/fs.h>
2730008 #include <kernel/lib_k.h>
2730009 //-------------------------------------------------------
2730010 #define DEBUG 0
2730011 //-------------------------------------------------------
2730012 int
2730013 tcp_connect (void *sock_item)
2730014 {
2730015   sock_t *sock = sock_item;
2730016   unsigned int lseq;
2730017   //
2730018   // Is there already a connection?
2730019   //
2730020   if (sock->tcp.conn == 0 || sock->tcp.conn == TCP_CLOSE)
2730021     {
2730022       //
2730023       // There isn't.
2730024       //
2730025       memset (sock->tcp.lsq, 0x00, sizeof (sock->tcp.lsq));
2730026       memset (sock->tcp.rsq, 0x00, sizeof (sock->tcp.rsq));
2730027       srand ((unsigned int) s_clock ((pid_t) 0));
2730028       lseq = rand ();
2730029       sock->tcp.lsq_ack = lseq + 1;
2730030       sock->tcp.lsq[++sock->tcp.lsqi] = lseq;
2730031       //
2730032       sock->tcp.can_send = 1;
2730033       sock->tcp.send_size = 0;
2730034       sock->tcp.send_flags = TCP_FLAG_SYN;
2730035       tcp_tx_sock (sock);
```

```
2730036         //
2730037         sock->tcp.conn = TCP_SYN_SENT;
2730038         //
2730039         // The operation has begun and will take some
2730040         // time.
2730041         //
2730042         errset (EINPROGRESS);
2730043         return (-1);
2730044       }
2730045     else if (sock->tcp.conn == TCP_RESET)
2730046       {
2730047         //
2730048         // Cannot connect: the socket status is reset to
2730049         // closed, to let
2730050         // the process retry, if it is really willing to
2730051         // do it.
2730052         //
2730053         sock->tcp.conn = TCP_CLOSE;
2730054         errset (ECONNREFUSED);
2730055         return (-1);
2730056       }
2730057     else if (sock->tcp.conn == TCP_SYN_SENT
2730058              || sock->tcp.conn == TCP_SYN_RECV)
2730059       {
2730060         //
2730061         // Already in progress.
2730062         //
2730063         errset (EALREADY);
2730064         return (-1);
2730065       }
2730066     else if (sock->tcp.conn == TCP_ESTABLISHED)
2730067       {
2730068         //
2730069         // Connection established.
2730070         //
2730071         return (0);
2730072       }
2730073     else
2730074       {
2730075         //
2730076         // Cannot reconnect.
2730077         //
2730078         errset (EISCONN);
2730079         return (-1);
2730080       }
2730081 }
```

94.12.44 kernel/net/tcp/tcp_rx_ack.c

«

Si veda la sezione 93.23.

```
2740001 #include <kernel/net.h>
2740002 #include <kernel/net/ip.h>
2740003 #include <kernel/net/route.h>
2740004 #include <kernel/net/tcp.h>
2740005 #include <sys/os32.h>
2740006 #include <kernel/lib_k.h>
2740007 #include <kernel/fs.h>
2740008 #include <errno.h>
2740009 #include <arpa/inet.h>
2740010 #include <netinet/in.h>
2740011 #include <netinet/tcp.h>
2740012 //-----------------------------------------------------------
2740013 #define DEBUG 0
2740014 //-----------------------------------------------------------
2740015 int
2740016 tcp_rx_ack (void *sock_item, void *packet)
2740017 {
2740018   sock_t *sock = sock_item;
2740019   struct iphdr *iphdr = packet;
2740020   struct tcphdr *tcphdr = (struct tcphdr *)
2740021     &((uint8_t *) packet)[iphdr->ihl * 4];
2740022   int i;
2740023   //
2740024   // Is the ACK sequence right?
2740025   //
2740026   if (sock->tcp.lsq_ack != ntohl (tcphdr->ack_seq))
2740027     {
2740028       //
2740029       // If it is a previous sequence, just ignore
2740030       // the packet.
2740031       //
2740032       for (i = 0; i < 16; i++)
2740033         {
2740034           if (sock->tcp.lsq[i] == ntohl (tcphdr->ack_seq))
2740035             {
2740036               break;
```



```
2740037             }
2740038         }
2740039       if (i >= 16)
2740040         {
2740041           if (DEBUG)
2740042             {
2740043               k_printf ("ERR loc seq: ");
2740044               tcp_show (ntohl (iphdr->saddr),
2740045                         ntohl (iphdr->daddr), tcphdr);
2740046
2740047               int j;
2740048               for (j = 0; j < 16; j++)
2740049                 {
2740050                   if (sock->tcp.lsq[j] != 0)
2740051                     {
2740052                       k_printf ("%3u ", sock->tcp.lsq[j]);
2740053                     }
2740054                 }
2740055               k_printf ("lsq_ack=%3u ", sock->tcp.lsq_ack);
2740056               k_printf ("\n");
2740057             }
2740058           //
2740059           // The ACK is out of sequence: sorry.
2740060           //
2740061           tcp_tx_rst (iphdr);
2740062         }
2740063       return (-1);
2740064     }
2740065   //
2740066   // Is the TCP sequence what we expected? But notice
2740067   // that, if our
2740068   // remote expected sequence is zero, this is the
2740069   // first time that
2740070   // get it, so it is right.
2740071   //
2740072   if (sock->tcp.rsq[sock->tcp.rsqi] != 0
2740073       && sock->tcp.rsq[sock->tcp.rsqi] !=
2740074       ntohl (tcphdr->seq))
2740075     {
2740076       //
2740077       // If it is a previous sequence, just ignore
2740078       // the packet.
2740079       //
2740080       for (i = 0; i < 16; i++)
2740081         {
2740082           if (sock->tcp.rsq[i] == ntohl (tcphdr->seq))
2740083             {
2740084               break;
2740085             }
2740086         }
2740087       if (i >= 16)
2740088         {
2740089           //
2740090           // The packet is out of sequence: sorry.
2740091           //
2740092           if (DEBUG)
2740093             {
2740094               k_printf ("ERR rem seq: ");
2740095               tcp_show (ntohl (iphdr->saddr),
2740096                         ntohl (iphdr->daddr), tcphdr);
2740097
2740098               int j;
2740099               for (j = 0; j < 16; j++)
2740100                 {
2740101                   if (sock->tcp.rsq[j] != 0)
2740102                     {
2740103                       k_printf ("%3u ", sock->tcp.rsq[j]);
2740104                     }
2740105                 }
2740106               k_printf ("\n");
2740107             }
2740108           tcp_tx_rst (iphdr);
2740109         }
2740110       return (-1);
2740111     }
2740112   return (0);
2740113 }
```

94.12.45 kernel/net/tcp/tcp_rx_data.c

«

Si veda la sezione 93.23.

```
2750001 #include <kernel/net.h>
2750002 #include <kernel/net/ip.h>
2750003 #include <kernel/net/route.h>
2750004 #include <kernel/net/tcp.h>
2750005 #include <sys/os32.h>
```

```
2750006 #include <kernel/lib_k.h>
2750007 #include <kernel/fs.h>
2750008 #include <errno.h>
2750009 #include <arpa/inet.h>
2750010 #include <netinet/in.h>
2750011 #include <netinet/tcp.h>
2750012 //-------------------------------------------------------
2750013 #define DEBUG 0
2750014 //-------------------------------------------------------
2750015 int
2750016 tcp_rx_data (void *sock_item, void *packet)
2750017 {
2750018   sock_t *sock = sock_item;
2750019   struct iphdr *iphdr = packet;
2750020   struct tcphdr *tcphdr = (struct tcphdr *)
2750021     &((uint8_t *) packet)[iphdr->ihl * 4];
2750022   uint8_t *recv_data;
2750023   size_t recv_size;
2750024   //
2750025   recv_data = &((uint8_t *) tcphdr)[tcphdr->doff * 4];
2750026   recv_size = ntohs (iphdr->tot_len) - (iphdr->ihl * 4)
2750027     - (tcphdr->doff * 4);
2750028   //
2750029   if (DEBUG)
2750030     {
2750031       if (recv_size > 0 && !sock->tcp.can_recv)
2750032         {
2750033           k_printf ("[%s] not ready to get data\n",
2750034                     __func__);
2750035         }
2750036     }
2750037   //
2750038   // If we receive zero data, it is ok.
2750039   //
2750040   if (recv_size == 0)
2750041     {
2750042       //
2750043       // Nothing to do, but there is no error.
2750044       //
2750045       return (0);
2750046     }
2750047   if (recv_size < 0)
2750048     {
2750049       return (-1);
2750050     }
2750051   if (recv_size > 0 && !sock->tcp.can_recv)
2750052     {
2750053       return (-1);
2750054     }
2750055   //
2750056   // Check the size.
2750057   //
2750058   if (recv_size > sizeof (sock->tcp.recv_data))
2750059     {
2750060       k_printf ("[%s] cannot accept a packet "
2750061                 "payload of %u bytes; packet "
2750062                 "truncated at %u bytes!\n",
2750063                 __func__, recv_size,
2750064                 sizeof (sock->tcp.recv_data));
2750065       //
2750066       recv_size = sizeof (sock->tcp.recv_data);
2750067     }
2750068   //
2750069   memcpy (sock->tcp.recv_data, recv_data, recv_size);
2750070   sock->tcp.recv_size = recv_size;
2750071   sock->tcp.recv_index = sock->tcp.recv_data;
2750072   //
2750073   // Must ACK back for the data received.
2750074   //
2750075   // The remote sequence that we will expect next time
2750076   // and the local sequence, expected from the other
2750077   // side.
2750078   //
2750079   sock->tcp.rsq[++sock->tcp.rsqi] =
2750080     ntohl (tcphdr->seq) + recv_size;
2750081   sock->tcp.lsq[++sock->tcp.lsqi] = sock->tcp.lsq_ack;
2750082   //
2750083   tcp_tx_ack (sock);
2750084   //
2750085   // Now the received data, if any, is to be read.
2750086   //
2750087   sock->tcp.can_recv = 0;
2750088   sock->tcp.can_read = 1;
2750089   //
2750090   //
2750091   //
2750092   return (0);
2750093 }
```


94.12.46 kernel/net/tcp/tcp_show.c

Si veda la sezione 93.23.

```
2760001 #include <kernel/net.h>
2760002 #include <kernel/net/ip.h>
2760003 #include <kernel/net/route.h>
2760004 #include <kernel/net/tcp.h>
2760005 #include <sys/os32.h>
2760006 #include <kernel/lib_k.h>
2760007 #include <kernel/fs.h>
2760008 #include <errno.h>
2760009 #include <arpa/inet.h>
2760010 #include <netinet/in.h>
2760011 #include <netinet/tcp.h>
2760012 //-------------------------------------------------------
2760013 void
2760014 tcp_show (h_addr_t src, h_addr_t dst,
2760015           const struct tcphdr *tcphdr)
2760016 {
2760017   struct in_addr addr_1;
2760018   struct in_addr addr_2;
2760019   char addr_string_1[INET_ADDRSTRLEN];
2760020   char addr_string_2[INET_ADDRSTRLEN];
2760021   //
2760022   if (tcphdr == NULL)
2760023     {
2760024       return;
2760025     }
2760026   //
2760027   addr_1.s_addr = htonl (src);
2760028   addr_2.s_addr = htonl (dst);
2760029   inet_ntop (AF_INET, &addr_1, addr_string_1,
2760030              (socklen_t) sizeof (addr_string_1));
2760031   inet_ntop (AF_INET, &addr_2, addr_string_2,
2760032              (socklen_t) sizeof (addr_string_2));
2760033   k_printf ("TCP %s:%i > %s:%i ", addr_string_1,
2760034             (unsigned int) ntohs (tcphdr->source),
2760035             addr_string_2,
2760036             (unsigned int) ntohs (tcphdr->dest));
2760037   k_printf ("s=%3u ", ntohl (tcphdr->seq));
2760038   k_printf ("k=%3u ", ntohl (tcphdr->ack_seq));
2760039
2760040   if (tcphdr->ack)
2760041     k_printf ("ack ");
2760042   if (tcphdr->psh)
2760043     k_printf ("psh ");
2760044   if (tcphdr->rst)
2760045     k_printf ("rst ");
2760046   if (tcphdr->syn)
2760047     k_printf ("syn ");
2760048   if (tcphdr->fin)
2760049     k_printf ("fin ");
2760050
2760051   k_printf ("\n");
2760052 }
```

94.12.47 kernel/net/tcp/tcp_status.c

Si veda la sezione 93.23.

```
2770001 #include <kernel/net.h>
2770002 #include <kernel/net/ip.h>
2770003 #include <kernel/net/route.h>
2770004 #include <kernel/net/tcp.h>
2770005 #include <sys/os32.h>
2770006 #include <kernel/lib_k.h>
2770007 #include <kernel/fs.h>
2770008 #include <errno.h>
2770009 #include <stdlib.h>
2770010 #include <arpa/inet.h>
2770011 #include <netinet/in.h>
2770012 #include <netinet/tcp.h>
2770013 //-------------------------------------------------------
2770014 int
2770015 tcp_status (void *ip_packet)
2770016 {
2770017   struct iphdr *iphdr;
2770018   struct tcphdr *tcphdr;
2770019   int s;
2770020   //
2770021   if (ip_packet == NULL)
2770022     {
2770023       errset (EINVAL);
2770024       return (-1);
```

```
2770025       }
2770026     //
2770027     iphdr = ip_packet;
2770028     tcphdr = (struct tcphdr *)
2770029       &(((uint8_t *) ip_packet)[iphdr->ihl * 4]);
2770030     //
2770031     if (iphdr->saddr == 0 || iphdr->daddr == 0)
2770032       {
2770033         errset (EINVAL);
2770034         return (-1);
2770035       }
2770036     //
2770037     if (tcphdr->source == 0 || tcphdr->dest == 0)
2770038       {
2770039         errset (EINVAL);
2770040         return (-1);
2770041       }
2770042     //
2770043     // Find a connection with the same IPs and ports.
2770044     //
2770045     for (s = 0; s < SOCK_MAX_SLOTS; s++)
2770046       {
2770047         if (!sock_table[s].active)
2770048           continue;
2770049         if (sock_table[s].family != AF_INET)
2770050           continue;
2770051         if (sock_table[s].protocol != IPPROTO_TCP)
2770052           continue;
2770053         if (sock_table[s].unreach_port)
2770054           continue;
2770055         if (sock_table[s].unreach_host)
2770056           continue;
2770057         if (sock_table[s].laddr != ntohl (iphdr->daddr))
2770058           continue;
2770059         if (sock_table[s].lport != ntohs (tcphdr->dest))
2770060           continue;
2770061         if (sock_table[s].raddr != ntohl (iphdr->saddr))
2770062           continue;
2770063         if (sock_table[s].rport != ntohs (tcphdr->source))
2770064           continue;
2770065         //
2770066         // A corresponding socket was found.
2770067         //
2770068         return ((int) sock_table[s].tcp.conn);
2770069       }
2770070     //
2770071     // Socket not found.
2770072     //
2770073     return (0);
2770074 }
```

94.12.48 kernel/net/tcp/tcp_test.c

«

Si veda la sezione 93.23.

```
2780001 #include <kernel/driver/pci.h>
2780002 #include <kernel/net/ip.h>
2780003 #include <kernel/net/tcp.h>
2780004 #include <kernel/net.h>
2780005 #include <kernel/ibm_i386.h>
2780006 #include <errno.h>
2780007 #include <kernel/lib_k.h>
2780008 #include <kernel/lib_s.h>
2780009 #include <stdint.h>
2780010 //-------------------------------------------------------
2780011 void
2780012 tcp_test (void)
2780013 {
2780014
2780015   h_addr_t src = 0xAC150B10;    // 172, 21, 11, 16
2780016   h_addr_t dst = 0xAC150B0F;    // 172, 21, 11, 15
2780017
2780018   int status;
2780019   //
2780020   //
2780021   //
2780022   status = tcp_tx_raw (12345, 1234, 100000, 0,
2780023                        TCP_FLAG_SYN,
2780024                        src, dst, "SYN", (size_t) 4);
2780025
2780026   if (status)
2780027     {
2780028       k_perror (NULL);
2780029     }
2780030 }
```


94.12.49 kernel/net/tcp/tcp_tx_ack.c

«

Si veda la sezione 93.23.

```
2790001 #include <kernel/net.h>
2790002 #include <kernel/net/ip.h>
2790003 #include <kernel/net/route.h>
2790004 #include <kernel/net/tcp.h>
2790005 #include <sys/os32.h>
2790006 #include <kernel/lib_k.h>
2790007 #include <kernel/fs.h>
2790008 #include <errno.h>
2790009 #include <arpa/inet.h>
2790010 #include <netinet/in.h>
2790011 #include <netinet/tcp.h>
2790012 //-------------------------------------------------------
2790013 #define DEBUG 0
2790014 //-------------------------------------------------------
2790015 int
2790016 tcp_tx_ack (void *sock_item)
2790017 {
2790018   sock_t *sock = sock_item;
2790019   tcp_packet_t packet;
2790020   tcp_pseudo_header_t pseudo;
2790021   uint16_t checksum;
2790022   //
2790023   if (sock == NULL)
2790024     {
2790025       errset (EINVAL);
2790026       return (-1);
2790027     }
2790028   if (sock->laddr == 0 || sock->raddr == 0)
2790029     {
2790030       errset (EINVAL);
2790031       return (-1);
2790032     }
2790033   if (sock->lport == 0 || sock->rport == 0)
2790034     {
2790035       errset (EINVAL);
2790036       return (-1);
2790037     }
2790038   //
2790039   // Prepare the TCP packet.
2790040   //
2790041   memset (&packet.header, 0, sizeof (struct tcphdr));
2790042   //
2790043   packet.header.source = htons (sock->lport);
2790044   packet.header.dest = htons (sock->rport);
2790045   packet.header.seq = htonl (sock->tcp.lsq[sock->tcp.lsqi]);
2790046   packet.header.ack_seq =
2790047     htonl (sock->tcp.rsq[sock->tcp.rsqi]);
2790048   packet.header.doff = (sizeof (struct tcphdr) / 4);
2790049   packet.header.ack = 1;
2790050   packet.header.window = htons (TCP_MSS);       // Minimal
2790051   // window
2790052   packet.header.check = 0;
2790053   //
2790054   // Prepare the pseudo header.
2790055   //
2790056   pseudo.saddr = htonl (sock->laddr);
2790057   pseudo.daddr = htonl (sock->raddr);
2790058   pseudo.zero = 0;
2790059   pseudo.protocol = IPPROTO_TCP;
2790060   pseudo.length = htons (sizeof (struct tcphdr));
2790061   //
2790062   // Now set the header checksum.
2790063   //
2790064   checksum =
2790065     ~(ip_checksum
2790066       ((void *) &packet, sizeof (struct tcphdr),
2790067        (void *) &pseudo, sizeof (tcp_pseudo_header_t)));
2790068   if (checksum == 0)
2790069     {
2790070       checksum = 0xFFFF;
2790071     }
2790072   packet.header.check = htons (checksum);
2790073   //
2790074   // Send to the lower network level.
2790075   //
2790076   if (DEBUG)
2790077     {
2790078       tcp_show (sock->laddr, sock->raddr,
2790079                 (struct tcphdr *) &packet);
2790080     }
2790081   return (ip_tx
2790082           (sock->laddr, sock->raddr, (int) IPPROTO_TCP,
2790083            &packet, sizeof (struct tcphdr)));
2790084 }
```

94.12.50 kernel/net/tcp/tcp_tx_raw.c

Si veda la sezione 93.23.

```
2800001 #include <kernel/net.h>
2800002 #include <kernel/net/ip.h>
2800003 #include <kernel/net/route.h>
2800004 #include <kernel/net/tcp.h>
2800005 #include <sys/os32.h>
2800006 #include <kernel/lib_k.h>
2800007 #include <errno.h>
2800008 #include <arpa/inet.h>
2800009 #include <netinet/tcp.h>
2800010 //-------------------------------------------------------
2800011 #define DEBUG 0
2800012 //-------------------------------------------------------
2800013 int
2800014 tcp_tx_raw (h_port_t sport, h_port_t dport,
2800015             uint32_t seq, uint32_t ack_seq, int flags,
2800016             h_addr_t saddr, h_addr_t daddr,
2800017             const void *buffer, size_t size)
2800018 {
2800019   tcp_packet_t packet;
2800020   tcp_pseudo_header_t pseudo;
2800021   uint16_t checksum;
2800022   size_t tcp_size = size + sizeof (struct tcphdr);
2800023   //
2800024   // Verify to have the source address: it is
2800025   // necessary here for
2800026   // the checksum calculation.
2800027   //
2800028   if (saddr == 0)
2800029     {
2800030       //
2800031       // Default source address: get the source
2800032       // address from the routing
2800033       // table, based on the destination.
2800034       //
2800035       saddr = route_remote_to_local (daddr);
2800036       if (saddr == ((h_addr_t) - 1))
2800037         {
2800038           errset (errno);
2800039           return (-1);
2800040         }
2800041     }
2800042   //
2800043   // Prepare the TCP packet.
2800044   //
2800045   memset (&packet.header, 0, sizeof (struct tcphdr));
2800046   //
2800047   packet.header.source = htons (sport);
2800048   packet.header.dest = htons (dport);
2800049   packet.header.seq = htonl (seq);
2800050   packet.header.ack_seq = htonl (ack_seq);
2800051   packet.header.doff = (sizeof (struct tcphdr) / 4);
2800052   if (flags & TCP_FLAG_ACK)
2800053     packet.header.ack = 1;
2800054   if (flags & TCP_FLAG_PSH)
2800055     packet.header.psh = 1;
2800056   if (flags & TCP_FLAG_RST)
2800057     packet.header.rst = 1;
2800058   if (flags & TCP_FLAG_SYN)
2800059     packet.header.syn = 1;
2800060   if (flags & TCP_FLAG_FIN)
2800061     packet.header.fin = 1;
2800062   if (flags & TCP_FLAG_RST)
2800063     {
2800064       packet.header.window = htons (0);
2800065     }
2800066   else
2800067     {
2800068       //
2800069       // Minimal window.
2800070       //
2800071       packet.header.window = htons (TCP_MSS);
2800072     }
2800073   packet.header.check = 0;
2800074   //
2800075   memcpy (packet.data, buffer, size);
2800076   //
2800077   // Prepare the pseudo header.
2800078   //
2800079   pseudo.saddr = htonl (saddr);
2800080   pseudo.daddr = htonl (daddr);
2800081   pseudo.zero = 0;
2800082   pseudo.protocol = IPPROTO_TCP;
2800083   pseudo.length = htons (tcp_size);
2800084   //
2800085   // Now set the header checksum.
```



```
2800086   //
2800087   checksum = ~(ip_checksum ((void *) &packet, tcp_size,
2800088                             (void *) &pseudo,
2800089                             sizeof (tcp_pseudo_header_t)));
2800090   if (checksum == 0)
2800091     {
2800092       checksum = 0xFFFF;
2800093     }
2800094   packet.header.check = htons (checksum);
2800095   //
2800096   // Send to the lower network level.
2800097   //
2800098   return (ip_tx
2800099           (saddr, daddr, (int) IPPROTO_TCP, &packet,
2800100            tcp_size));
2800101 }
```

94.12.51 kernel/net/tcp/tcp_tx_rst.c

Si veda la sezione 93.23.

```
2810001 #include <kernel/net.h>
2810002 #include <kernel/net/ip.h>
2810003 #include <kernel/net/route.h>
2810004 #include <kernel/net/tcp.h>
2810005 #include <sys/os32.h>
2810006 #include <kernel/lib_k.h>
2810007 #include <kernel/fs.h>
2810008 #include <errno.h>
2810009 #include <stdlib.h>
2810010 #include <arpa/inet.h>
2810011 #include <netinet/in.h>
2810012 #include <netinet/tcp.h>
2810013 //-------------------------------------------------------
2810014 #define DEBUG 0
2810015 //-------------------------------------------------------
2810016 int
2810017 tcp_tx_rst (void *ip_packet)
2810018 {
2810019   struct iphdr *iphdr = ip_packet;
2810020   struct tcphdr *tcphdr;
2810021   uint32_t seq;
2810022   uint32_t ack_seq;
2810023   tcp_packet_t packet;
2810024   tcp_pseudo_header_t pseudo;
2810025   uint16_t checksum;
2810026   //
2810027   if (ip_packet == NULL)
2810028     {
2810029       errset (EINVAL);
2810030       return (-1);
2810031     }
2810032   //
2810033   iphdr = ip_packet;
2810034   tcphdr = (struct tcphdr *)
2810035     &(((uint8_t *) ip_packet)[iphdr->ihl * 4]);
2810036   //
2810037   if (iphdr->saddr == 0 || iphdr->daddr == 0)
2810038     {
2810039       errset (EINVAL);
2810040       return (-1);
2810041     }
2810042   //
2810043   if (tcphdr->source == 0 || tcphdr->dest == 0)
2810044     {
2810045       errset (EINVAL);
2810046       return (-1);
2810047     }
2810048   //
2810049   // If the bad TCP packet has a ACK sequence, we
2810050   // replay with
2810051   // the same sequence (the one that the other side
2810052   // expects).
2810053   //
2810054   if (tcphdr->ack)
2810055     {
2810056       seq = ntohl (tcphdr->ack_seq);
2810057     }
2810058   else
2810059     {
2810060       seq = rand ();
2810061     }
2810062   //
2810063   // Our reset ACK has the same sequence received.
2810064   //
2810065   ack_seq = ntohl (tcphdr->seq);
2810066   //
```

```
2810067   // Prepare the TCP packet.
2810068   //
2810069   memset (&packet.header, 0, sizeof (struct tcphdr));
2810070   //
2810071   packet.header.source = tcphdr->dest;
2810072   packet.header.dest = tcphdr->source;
2810073   packet.header.seq = htonl (seq);
2810074   packet.header.ack_seq = htonl (ack_seq);
2810075   packet.header.doff = (sizeof (struct tcphdr) / 4);
2810076   packet.header.ack = 1;
2810077   packet.header.rst = 1;
2810078   packet.header.window = 0;
2810079   packet.header.check = 0;
2810080   //
2810081   // Prepare the pseudo header.
2810082   //
2810083   pseudo.saddr = iphdr->saddr;
2810084   pseudo.daddr = iphdr->daddr;
2810085   pseudo.zero = 0;
2810086   pseudo.protocol = IPPROTO_TCP;
2810087   pseudo.length = htons (sizeof (struct tcphdr));
2810088   //
2810089   // Now set the header checksum.
2810090   //
2810091   checksum =
2810092     ~(ip_checksum
2810093       ((void *) &packet, sizeof (struct tcphdr),
2810094        (void *) &pseudo, sizeof (tcp_pseudo_header_t)));
2810095   if (checksum == 0)
2810096     {
2810097       checksum = 0xFFFF;
2810098     }
2810099   packet.header.check = htons (checksum);
2810100   //
2810101   // Send to the lower network level.
2810102   //
2810103   if (DEBUG)
2810104     {
2810105       tcp_show (ntohl (iphdr->saddr),
2810106                 ntohl (iphdr->daddr),
2810107                 (struct tcphdr *) &packet);
2810108     }
2810109   return (ip_tx
2810110           (ntohl (iphdr->saddr), ntohl (iphdr->daddr),
2810111            IPPROTO_TCP, &packet, sizeof (struct tcphdr)));
2810112 }
```

94.12.52 kernel/net/tcp/tcp_tx_sock.c

«

Si veda la sezione 93.23.

```
2820001 #include <kernel/net.h>
2820002 #include <kernel/net/ip.h>
2820003 #include <kernel/net/route.h>
2820004 #include <kernel/net/tcp.h>
2820005 #include <sys/os32.h>
2820006 #include <kernel/lib_k.h>
2820007 #include <kernel/fs.h>
2820008 #include <errno.h>
2820009 #include <arpa/inet.h>
2820010 #include <netinet/in.h>
2820011 #include <netinet/tcp.h>
2820012 //-----------------------------------------------------------
2820013 #define DEBUG 0
2820014 //-----------------------------------------------------------
2820015 int
2820016 tcp_tx_sock (void *sock_item)
2820017 {
2820018   sock_t *sock = sock_item;
2820019   tcp_packet_t packet;
2820020   tcp_pseudo_header_t pseudo;
2820021   uint16_t checksum;
2820022   size_t tcp_size;
2820023   uint32_t seq;
2820024   uint32_t ack_seq;
2820025   size_t send_size;
2820026   //
2820027   if (sock == NULL)
2820028     {
2820029       errset (EINVAL);
2820030       return (-1);
2820031     }
2820032   if (!sock->tcp.can_send)
2820033     {
2820034       errset (EINVAL);
2820035       return (-1);
2820036     }
```



```
2820037   if (sock->laddr == 0 || sock->raddr == 0)
2820038     {
2820039       errset (EINVAL);
2820040       return (-1);
2820041     }
2820042   if (sock->lport == 0 || sock->rport == 0)
2820043     {
2820044       errset (EINVAL);
2820045       return (-1);
2820046     }
2820047   //
2820048   // Sequences and size.
2820049   //
2820050   seq = sock->tcp.lsq[sock->tcp.lsqi];
2820051   if ((sock->tcp.send_flags & TCP_FLAG_SYN)
2820052       || (sock->tcp.send_flags & TCP_FLAG_FIN))
2820053     {
2820054       //
2820055       // A SYN or FIN packet cannot load data.
2820056       //
2820057       send_size = 0;
2820058       //
2820059       // The next expected ACK from the other side is
2820060       // just
2820061       // +1.
2820062       //
2820063       sock->tcp.lsq_ack = seq + 1;
2820064     }
2820065   else
2820066     {
2820067       send_size = sock->tcp.send_size;
2820068       //
2820069       // The next expected ACK from the other side is
2820070       // + size of the sent data.
2820071       //
2820072       sock->tcp.lsq_ack = seq + send_size;
2820073     }
2820074   //
2820075   if (sock->tcp.send_flags & TCP_FLAG_ACK)
2820076     {
2820077       ack_seq = sock->tcp.rsq[sock->tcp.rsqi];
2820078     }
2820079   else
2820080     {
2820081       ack_seq = 0;
2820082     }
2820083   //
2820084   // Prepare the TCP packet.
2820085   //
2820086   memset (&packet.header, 0, sizeof (struct tcphdr));
2820087   //
2820088   packet.header.source = htons (sock->lport);
2820089   packet.header.dest = htons (sock->rport);
2820090   packet.header.seq = htonl (seq);
2820091   packet.header.ack_seq = htonl (ack_seq);
2820092   packet.header.doff = (sizeof (struct tcphdr) / 4);
2820093   if (sock->tcp.send_flags & TCP_FLAG_ACK)
2820094     packet.header.ack = 1;
2820095   if (sock->tcp.send_flags & TCP_FLAG_PSH)
2820096     packet.header.psh = 1;
2820097   if (sock->tcp.send_flags & TCP_FLAG_RST)
2820098     packet.header.rst = 1;
2820099   if (sock->tcp.send_flags & TCP_FLAG_SYN)
2820100     packet.header.syn = 1;
2820101   if (sock->tcp.send_flags & TCP_FLAG_FIN)
2820102     packet.header.fin = 1;
2820103   if (sock->tcp.send_flags & TCP_FLAG_RST)
2820104     {
2820105       packet.header.window = htons (0);
2820106     }
2820107   else
2820108     {
2820109       //
2820110       // Minimal window.
2820111       //
2820112       packet.header.window = htons (TCP_MSS);
2820113     }
2820114   packet.header.check = 0;
2820115   //
2820116   memcpy (packet.data, sock->tcp.send_data, send_size);
2820117   //
2820118   tcp_size = sizeof (struct tcphdr) + send_size;
2820119   //
2820120   // Prepare the pseudo header.
2820121   //
2820122   pseudo.saddr = htonl (sock->laddr);
2820123   pseudo.daddr = htonl (sock->raddr);
```

```
2820124   pseudo.zero = 0;
2820125   pseudo.protocol = IPPROTO_TCP;
2820126   pseudo.length = htons (tcp_size);
2820127   //
2820128   // Now set the header checksum.
2820129   //
2820130   checksum = ~(ip_checksum ((void *) &packet, tcp_size,
2820131                             (void *) &pseudo,
2820132                             sizeof (tcp_pseudo_header_t)));
2820133   if (checksum == 0)
2820134     {
2820135       checksum = 0xFFFF;
2820136     }
2820137   packet.header.check = htons (checksum);
2820138   //
2820139   // Send to the lower network level.
2820140   //
2820141   if (DEBUG)
2820142     {
2820143       tcp_show (sock->laddr, sock->raddr,
2820144                 (struct tcphdr *) &packet);
2820145     }
2820146   sock->tcp.clock = s_clock ((pid_t) 0);
2820147   return (ip_tx
2820148           (sock->laddr, sock->raddr, (int) IPPROTO_TCP,
2820149            &packet, tcp_size));
2820150 }
```

94.12.53 kernel/net/udp.h

Si veda la sezione 93.23.

```
2830001 #ifndef _KERNEL_NET_UDP_H
2830002 #define _KERNEL_NET_UDP_H    1
2830003 //-------------------------------------------------------------
2830004 #include <netinet/udp.h>
2830005 #include <kernel/net.h>
2830006 //-------------------------------------------------------------
2830007 #define UDP_HEADER_SIZE  8
2830008 #define UDP_MAX_PACKET_SIZE  NET_IP_MAX_DATA_SIZE
2830009 #define UDP_MAX_DATA_SIZE    \
2830010     UDP_MAX_PACKET_SIZE-UDP_HEADER_SIZE
2830011 //-------------------------------------------------------------
2830012 //
2830013 // UDP packet, for transmission.
2830014 //
2830015 typedef struct
2830016 {
2830017   struct udphdr header;
2830018   uint8_t data[UDP_MAX_DATA_SIZE];
2830019 } __attribute__ ((packed)) udp_packet_t;
2830020 //
2830021 // UDP pseudo header for checksum calculation.
2830022 //
2830023 typedef struct
2830024 {
2830025   in_addr_t saddr;
2830026   in_addr_t daddr;
2830027   uint8_t zero;
2830028   uint8_t protocol;
2830029   uint16_t length;
2830030 } __attribute__ ((packed)) udp_pseudo_header_t;
2830031 //-------------------------------------------------------------
2830032 int udp_tx (h_port_t sport, h_port_t dport,
2830033             h_addr_t saddr, h_addr_t daddr,
2830034             const void *buffer, size_t size);
2830035 //-------------------------------------------------------------
2830036 #endif
```

94.12.54 kernel/net/udp/udp_tx.c

Si veda la sezione 93.23.

```
2840001 #include <kernel/net.h>
2840002 #include <kernel/net/ip.h>
2840003 #include <kernel/net/route.h>
2840004 #include <kernel/net/udp.h>
2840005 #include <sys/os32.h>
2840006 #include <kernel/lib_k.h>
2840007 #include <errno.h>
2840008 #include <arpa/inet.h>
2840009 #include <netinet/udp.h>
2840010 //-------------------------------------------------------------
2840011 #define DEBUG 0
2840012 //-------------------------------------------------------------
2840013 int
2840014 udp_tx (h_port_t sport, h_port_t dport, h_addr_t saddr,
2840015         h_addr_t daddr, const void *buffer, size_t size)
2840016 {
2840017   udp_packet_t packet;
2840018   udp_pseudo_header_t pseudo;
2840019   uint16_t checksum;
2840020   size_t udp_size = size + sizeof (struct udphdr);
2840021   //
2840022   // Verify to have the source address: it is
2840023   // necessary here for
2840024   // the checksum calculation.
2840025   //
2840026   if (saddr == 0)
2840027     {
2840028       //
2840029       // Default source address: get the source
2840030       // address from the routing
2840031       // table, based on the destination.
2840032       //
2840033       saddr = route_remote_to_local (daddr);
2840034       if (saddr == ((h_addr_t) - 1))
2840035         {
2840036           errset (errno);
2840037           return (-1);
2840038         }
2840039     }
2840040   //
2840041   // Prepare the UDP packet.
2840042   //
2840043   packet.header.source = htons (sport);
2840044   packet.header.dest = htons (dport);
2840045   packet.header.len = htons (udp_size);
2840046   packet.header.check = 0;
2840047   memcpy (packet.data, buffer, size);
2840048   //
2840049   // Prepare the pseudo header.
2840050   //
2840051   pseudo.saddr = htonl (saddr);
2840052   pseudo.daddr = htonl (daddr);
2840053   pseudo.zero = 0;
2840054   pseudo.protocol = IPPROTO_UDP;
2840055   pseudo.length = htons (udp_size);
2840056   //
2840057   // Now set the header checksum.
2840058   //
2840059   checksum = ~(ip_checksum ((void *) &packet, udp_size,
2840060                             (void *) &pseudo,
2840061                             sizeof (udp_pseudo_header_t)));
2840062   if (checksum == 0)
2840063     {
2840064       checksum = 0xFFFF;
2840065     }
2840066   packet.header.check = htons (checksum);
2840067   //
2840068   // Send to the lower network level.
2840069   //
2840070   return (ip_tx
2840071           (saddr, daddr, (int) IPPROTO_UDP, &packet,
2840072            udp_size));
2840073 }
```

94.13 os32: «kernel/part.h»

Si veda la sezione 93.18.

```
2850001 #ifndef _KERNEL_PART_H
2850002 #define _KERNEL_PART_H    1
2850003
2850004
2850005 typedef struct
2850006 {
2850007   uint8_t active;       // boot indicator 0 or
2850008   // PART_ACTIVE
2850009   uint8_t h_start;      // head value for first sector
2850010   uint8_t s_start;      // sector value + cyl bits for
2850011   // first sector
2850012   uint8_t c_start;      // track value for first
2850013   // sector
2850014   uint8_t type; // partition type
2850015   uint8_t h_last;       // head value for last sector
2850016   uint8_t s_last;       // sector value + cyl bits for
2850017   // last sector
2850018   uint8_t c_last;       // track value for last sector
2850019   uint32_t l_start;     // logical first sector
2850020   uint32_t size;        // size of partition in
2850021   // sectors
2850022 } part_t;
2850023
```

```
#define PART_ACTIVE     0x80    // value for active
                                // partition
#define PART_MAX        4       // number of entries
                                // in partition table
#define PART_TABLE_OFF  0x1BE   // offset of part.
                                // table in boot
                                // sector

//
// Partition types.
//
#define PART_TYPE_NONE      0x00        // unused
                                        // entry
#define PART_TYPE_NO_PART   0xFF        // full device
#define PART_TYPE_MINIX     0x81        // Minix
                                        // partition
                                        // type
#define PART_TYPE_OLDMINIX  0x80        // Minix 1 old
                                        // partition
                                        // type
#define PART_TYPE_EXT       0x05        // extended
                                        // partition


#endif
```

94.14 os32: «kernel/proc.h»

«

Si veda la sezione 93.20.

```
#ifndef _KERNEL_PROC_H
#define _KERNEL_PROC_H  1
//-------------------------------------------------------------
#include <kernel/ibm_i386.h>
#include <kernel/dev.h>
#include <kernel/driver/tty.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <kernel/fs.h>
#include <sys/os32.h>
#include <stddef.h>
#include <stdint.h>
#include <time.h>
//-------------------------------------------------------------
#define CLOCK_FREQUENCY_DIVISOR \
    (3579545/3/CLOCKS_PER_SEC)  // [1]
//
// [1] Internal clock frequency is (3579545/3) Hz.
//     This value is divided by
//     CLOCK_FREQUENCY_DIVISOR, giving 100 Hz.
//     The divisor value if fixed, because the code
//     suppose that the clock frequency is 100 Hz!
//
//-------------------------------------------------------------
#define PROC_EMPTY                  0
#define PROC_CREATED                1
#define PROC_READY                  2
#define PROC_RUNNING                3
#define PROC_SLEEPING               4
#define PROC_ZOMBIE                 5
//-------------------------------------------------------------
#define MAGIC_OS32_APPL   0x6F7333326170706CLL  // [2]
//
// [2] os32appl
//
//-------------------------------------------------------------
#define PROCESS_MAX  ((GDT_ITEMS/2)-1)  // Process
                                        // slots.
#define MAX_SIGNALS  31 // Max signal number.
//
typedef struct
{
  pid_t ppid;   // Parent PID.
  pid_t pgrp;   // Process group ID.
  uid_t uid;    // Real user ID
  uid_t euid;   // Effective user ID.
  uid_t suid;   // Saved user ID.
  gid_t gid;    // Real group ID
  gid_t egid;   // Effective group ID.
  gid_t sgid;   // Saved group ID.
  dev_t device_tty;     // Controlling terminal.
  char path_cwd[PATH_MAX];
  // Working directory path.
  inode_t *inode_cwd;   // Working directory inode.
  int umask;    // File creation mask.
```



```
  unsigned long int sig_status; // Active signals.
  unsigned long int sig_ignore; // Signals to be
  // ignored.
  uintptr_t sig_handler[MAX_SIGNALS];   // Opt. sig.
  // handlers.
  uintptr_t sig_handler_wrapper;        // Special
  // wrapper.
  clock_t usage;        // Clock ticks CPU time usage.
  unsigned int status;
  int wakeup_events;    // Wake up for something.
  int wakeup_signal;    // Signal waited.
  unsigned int wakeup_timer;    // Seconds to wait
  // for.
  inode_t *wakeup_inode;        // Inode waited.
  sock_t *wakeup_sock;  // Socket waited.
  dev_t wakeup_dev;     // Device waited.
  addr_t address_text;
  size_t domain_text;
  addr_t address_data;
  size_t domain_data;
  size_t domain_stack;  // Included inside the data.
  size_t extra_data;    // Extra data for 'brk()'.
  uint32_t sp;
  int ret;
  char name[PATH_MAX];
  fd_t fd[FOPEN_MAX];
} proc_t;
//
extern proc_t proc_table[PROCESS_MAX];
//
// ATTENTION: THERE IS NO WAY TO KEEP THE STACK DOMAIN
//            IN A DIFFERENT ADDRESS SPACE THAN THE
//            OTHER DATA. There is a simple explanation
//            for such limitation: A POINTER TO DATA
//            INSIDE THE STACK, CANNOT BE REACHED IF IT
//            IS NOT INSIDE THE SAME ADDRESS SPACE!
//
//            For the same reason, data or stack,
//            cannot be moved (shifted) down, when more
//            space is needed, because previous
//            pointers would not work! So, to develop
//            an extensible 'malloc' area, such memory
//            is to be placed *after* the stack, moving
//            *all* the data segment if necessary.
//-------------------------------------------------------------
extern pid_t proc_current;
extern uint32_t proc_stack_pointer;
extern uint16_t proc_stack_segment_selector;
extern unsigned int proc_loops_per_clock;
//-------------------------------------------------------------
typedef struct
{
  uint32_t filler0;
  uint64_t magic;
  uint32_t data_offset;
  uint32_t etext;
  uint32_t edata;
  uint32_t ebss;
  uint32_t ssize;
} header_t;
//-------------------------------------------------------------
typedef struct
{
  uint32_t eax;
  uint32_t ecx;
  uint32_t edx;
  uint32_t ebx;
  uint32_t ebp;
  uint32_t esi;
  uint32_t edi;
  uint32_t ds;
  uint32_t es;
  uint32_t fs;
  uint32_t gs;
  uint32_t eip;
  uint32_t cs;
  uint32_t eflags;
} stack_t;
//-------------------------------------------------------------
void proc_init (void);
void proc_timer_init (clock_t freq);
void proc_delay (void);
void proc_scheduler (void);
void sysroutine (uint32_t syscallnr, uint32_t msg_off,
                 uint32_t msg_size);
//-------------------------------------------------------------
void *ptr (pid_t pid, void *p);
```

```
2860143 proc_t *proc_reference (pid_t pid);
2860144 void proc_print (void);
2860145 //-------------------------------------------------------
2860146 int proc_sys_exec (pid_t pid, const char *path,
2860147                    unsigned int argc, char *arg_data,
2860148                    unsigned int envc, char *env_data);
2860149 //-------------------------------------------------------
2860150 void proc_dump_memory (pid_t pid, addr_t address,
2860151                        size_t size, char *name);
2860152 void proc_available (pid_t pid);
2860153 void proc_sch_net (void);
2860154 void proc_sch_signals (void);
2860155 void proc_sch_terminals (void);
2860156 void proc_sch_timers (void);
2860157 void proc_sig_chld (pid_t parent, int sig);
2860158 void proc_sig_cont (pid_t pid, int sig);
2860159 void proc_sig_core (pid_t pid, int sig);
2860160 void proc_sig_handler (pid_t pid, int sig);
2860161 int proc_sig_ignore (pid_t pid, int sig);
2860162 void proc_sig_off (pid_t pid, int sig);
2860163 void proc_sig_on (pid_t pid, int sig);
2860164 int proc_sig_status (pid_t pid, int sig);
2860165 void proc_sig_stop (pid_t pid, int sig);
2860166 void proc_sig_term (pid_t pid, int sig);
2860167
2860168 void proc_wakeup_pipe_read (inode_t * inode);
2860169 void proc_wakeup_pipe_write (inode_t * inode);
2860170 void proc_wakeup_terminal (void);
2860171
2860172 #endif
```


94.14.1 kernel/proc/proc_available.c

Si veda la sezione 93.20.1.

```
2870001 #include <kernel/proc.h>
2870002 //-------------------------------------------------------
2870003 void
2870004 proc_available (pid_t pid)
2870005 {
2870006   proc_table[pid].ppid = (pid_t) - 1;
2870007   proc_table[pid].pgrp = (pid_t) - 1;
2870008   proc_table[pid].uid = (uid_t) - 1;
2870009   proc_table[pid].euid = (uid_t) - 1;
2870010   proc_table[pid].suid = (uid_t) - 1;
2870011   proc_table[pid].gid = (gid_t) - 1;
2870012   proc_table[pid].egid = (gid_t) - 1;
2870013   proc_table[pid].sgid = (gid_t) - 1;
2870014   proc_table[pid].sig_status = 0;
2870015   proc_table[pid].sig_ignore = 0;
2870016   proc_table[pid].usage = (clock_t) 0;
2870017   proc_table[pid].status = PROC_EMPTY;
2870018   proc_table[pid].wakeup_events = 0;
2870019   proc_table[pid].wakeup_signal = 0;
2870020   proc_table[pid].wakeup_timer = 0;
2870021   proc_table[pid].wakeup_inode = NULL;
2870022   proc_table[pid].address_text = (addr_t) 0;
2870023   proc_table[pid].domain_text = (size_t) 0;
2870024   proc_table[pid].address_data = (addr_t) 0;
2870025   proc_table[pid].domain_data = (size_t) 0;
2870026   proc_table[pid].domain_stack = (size_t) 0;
2870027   proc_table[pid].extra_data = (size_t) 0;
2870028   proc_table[pid].sp = (uint32_t) 0;
2870029   proc_table[pid].ret = 0;
2870030   proc_table[pid].inode_cwd = NULL;
2870031   proc_table[pid].path_cwd[0] = 0;
2870032   proc_table[pid].umask = 0;
2870033   proc_table[pid].name[0] = 0;
2870034 }
```

94.14.2 kernel/proc/proc_dump_memory.c

Si veda la sezione 93.20.2.

```
2880001 #include <kernel/proc.h>
2880002 #include <fcntl.h>
2880003 #include <kernel/lib_s.h>
2880004 #include <kernel/lib_k.h>
2880005 //-------------------------------------------------------
2880006 void
2880007 proc_dump_memory (pid_t pid, addr_t address,
2880008                   size_t size, char *name)
2880009 {
2880010   int fdn;
2880011   char buffer[BUFSIZ];
2880012   ssize_t size_written;
2880013   ssize_t size_written_total;
2880014   ssize_t size_read;
2880015   ssize_t size_read_total;
2880016   ssize_t size_total = 0;
2880017   //
2880018   // Dump the code segment to disk.
2880019   //
2880020   fdn =
2880021     s_open (pid, name, (O_WRONLY | O_CREAT | O_TRUNC),
2880022             (mode_t) (S_IFREG | 00644));
2880023   if (fdn < 0)
2880024     {
2880025       //
2880026       // There is a problem: just let it go.
2880027       //
2880028       k_perror (NULL);
2880029       return;
2880030     }
2880031   //
2880032   // Read the memory and write it to disk.
2880033   //
2880034   for (size_read = 0, size_read_total = 0;
2880035        size_read_total < size_total;
2880036        size_read_total += size_read, address += size_read)
2880037     {
2880038       size_read = dev_io ((pid_t) 0, DEV_MEM, DEV_READ,
2880039                           (off_t) address, buffer,
2880040                           (size_t) BUFSIZ, NULL);
2880041       //
2880042       for (size_written = 0, size_written_total = 0;
2880043            size_written_total < size_read;
2880044            size_written_total += size_written)
2880045         {
2880046           size_written = s_write
2880047             (pid, fdn,
2880048              &buffer
2880049              [size_written_total],
2880050              (size_t) (size_read - size_written_total));
2880051           //
```

```
2880052             if (size_written < 0)
2880053               {
2880054                 s_close (pid, fdn);
2880055                 return;
2880056               }
2880057           }
2880058       }
2880059     s_close (pid, fdn);
2880060   }
```

94.14.3 kernel/proc/proc_init.c

«

Si veda la sezione 93.20.3.

```
2890001  #include <kernel/ibm_i386.h>
2890002  #include <kernel/proc.h>
2890003  #include <kernel/lib_k.h>
2890004  #include <kernel/lib_s.h>
2890005  #include <string.h>
2890006  #include <kernel/multiboot.h>
2890007  #include <kernel/dm.h>
2890008  //-------------------------------------------------------------
2890009  extern uint32_t _k_start;
2890010  extern uint32_t _k_end;
2890011  extern uint32_t _k_text_end;
2890012  extern uint32_t _k_data_end;
2890013  extern uint32_t _k_bss_end;
2890014  extern uint32_t _k_stack_top;
2890015  extern uint32_t _k_stack_bottom;
2890016  //-------------------------------------------------------------
2890017  void
2890018  proc_init (void)
2890019  {
2890020    pid_t pid;
2890021    int fdn;      // File descriptor index;
2890022    inode_t *inode;
2890023    sb_t *sb;
2890024    clock_t time_start;
2890025    clock_t time_now;
2890026    clock_t time_elapsed;
2890027    unsigned long long int count;
2890028    uintptr_t stack_top = (uintptr_t) & _k_stack_top;
2890029    uintptr_t stack_bottom = (uintptr_t) & _k_stack_bottom;
2890030    int sig;
2890031    //
2890032    // Set up the GDT table.
2890033    //
2890034    gdt ();
2890035    //
2890036    // Set up timer.
2890037    //
2890038    proc_timer_init (CLOCKS_PER_SEC);
2890039    //
2890040    // Disable external interrupt.
2890041    //
2890042    cli ();
2890043    //
2890044    // Set up the IDT table.
2890045    //
2890046    idt ();
2890047    //
2890048    // Set all memory reference to some invalid data.
2890049    //
2890050    for (pid = 0; pid < PROCESS_MAX; pid++)
2890051      {
2890052        proc_available (pid);
2890053      }
2890054    //
2890055    // Set up the process table with the kernel.
2890056    //
2890057    proc_table[0].ppid = 0;
2890058    proc_table[0].pgrp = 0;
2890059    proc_table[0].uid = 0;
2890060    proc_table[0].euid = 0;
2890061    proc_table[0].suid = 0;
2890062    proc_table[0].gid = 0;
2890063    proc_table[0].egid = 0;
2890064    proc_table[0].sgid = 0;
2890065    proc_table[0].device_tty = DEV_UNDEFINED;
2890066    proc_table[0].sig_status = 0;
2890067    proc_table[0].sig_ignore = 0;
2890068    proc_table[0].usage = 0;
2890069    proc_table[0].status = PROC_RUNNING;
2890070    proc_table[0].wakeup_events = 0;
2890071    proc_table[0].wakeup_signal = 0;
2890072    proc_table[0].wakeup_timer = 0;
2890073    proc_table[0].wakeup_inode = NULL;
```



```
2890074    proc_table[0].address_text = 0;                    // [1]
2890075    proc_table[0].domain_text = (size_t) (&_k_end);    // [2]
2890076    proc_table[0].address_data = 0;                    // [2]
2890077    proc_table[0].domain_data = (size_t) 0;            // [2]
2890078    proc_table[0].domain_stack =
2890079      (size_t) (stack_bottom - stack_top);
2890080    proc_table[0].extra_data = (size_t) 0;
2890081    proc_table[0].sp = 0; // [3]
2890082    proc_table[0].ret = 0;
2890083    proc_table[0].umask = 0022;   // Default umask.
2890084    strncpy (proc_table[0].path_cwd, "/", PATH_MAX);
2890085    strncpy (proc_table[0].name, "os32 kernel", PATH_MAX);
2890086    //
2890087    // [1] The kernel text starts at 0x100000, that is,
2890088    // 1 Mibyte,
2890089    // but the code expect to start at that address,
2890090    // just like
2890091    // the space from address 0 to 0xFFFFF is anyway
2890092    // part
2890093    // of the kernel. If the kernel is forked, as it
2890094    // happens
2890095    // when the 'init' process is to be run, it is
2890096    // necessary
2890097    // to consider part of the kernel all the addresses
2890098    // starting
2890099    // from zero.
2890100    //
2890101    // [2] The kernel is a unique block, where text and
2890102    // data live
2890103    // together.
2890104    //
2890105    // [3] The saved stack pointer location will be set
2890106    // at the
2890107    // next interrupt, or system call. That is why the
2890108    // kernel
2890109    // must send a null system call at the beginning of
2890110    // its
2890111    // work.
2890112    // -----------------------------------------------
2890113    //
2890114    // Ensure to have a terminated string.
2890115    //
2890116    proc_table[0].name[PATH_MAX - 1] = 0;
2890117    //
2890118    // Reset file descriptors.
2890119    //
2890120    for (fdn = 0; fdn < OPEN_MAX; fdn++)
2890121      {
2890122        proc_table[0].fd[fdn].fl_flags = 0;
2890123        proc_table[0].fd[fdn].fd_flags = 0;
2890124        proc_table[0].fd[fdn].file = NULL;
2890125      }
2890126    //
2890127    // Reset 'sig_handler[]'.
2890128    //
2890129    for (sig = 0; sig < MAX_SIGNALS; sig++)
2890130      {
2890131        proc_table[0].sig_handler[sig] = (uintptr_t) NULL;
2890132      }
2890133    //
2890134    // Allocate memory for the kernel.
2890135    //
2890136    // The BIOS data area (BDA) and extra BIOS at the
2890137    // bottom
2890138    // of 640 Kibyte, is already, formally, included
2890139    // inside the
2890140    // kernel.
2890141    //
2890142    // The allocation for data has no effect here,
2890143    // because it is
2890144    // the same as the text.
2890145    //
2890146    mb_alloc (proc_table[0].address_text,
2890147              proc_table[0].domain_text);
2890148    mb_alloc (proc_table[0].address_data,
2890149              proc_table[0].domain_data);
2890150    //
2890151    // Enable and disable hardware interrupts (IRQ).
2890152    //
2890153    irq_on (0);   // timer.
2890154    irq_on (1);   // keyboard
2890155    irq_on (2);   // PIC2: keep it ON! [4]
2890156    irq_on (3);   //
2890157    irq_on (4);   //
2890158    irq_on (5);   //
2890159    irq_on (6);   //
2890160    irq_on (7);   //
```

```
2890161   irq_on (8);   //
2890162   irq_on (9);   //
2890163   irq_on (10);  //
2890164   irq_on (11);  //
2890165   irq_on (12);  //
2890166   irq_on (13);  //
2890167   irq_on (14);  //
2890168   irq_on (15);  //
2890169   //
2890170   // [4] IRQ 2 must be let working, because inside the
2890171   // file
2890172   // 'kernel/ibm_i386/isr_s', the IRQ 2 is used to
2890173   // reset
2890174   // properly PIC 1, so that, the IRQ from 8 to 15
2890175   // will
2890176   // reset only PIC 2.
2890177   // The problem is that any PIC must be reset exactly
2890178   // once, otherwise the system will lock.
2890179   //
2890180   //
2890181   // External interrupts activation.
2890182   //
2890183   sti ();
2890184   //
2890185   // Calculate how many times can be executed the
2890186   // following loop. This data will be used by
2890187   // 'k_sleep()', if the clock pulse does not work
2890188   // for some reason.
2890189   //
2890190   time_elapsed = 0;
2890191   count = 0;
2890192   time_start = s_clock ((pid_t) 0);
2890193   for (; time_elapsed < 10; count++)
2890194     {
2890195       time_now = s_clock ((pid_t) 0);
2890196       time_elapsed = time_now - time_start;
2890197     }
2890198   proc_loops_per_clock = count / 10;
2890199   //
2890200   // Set up data-memory devices:
2890201   // it works only after enabling interrupts.
2890202   //
2890203   dm_init ();
2890204   //
2890205   // Mount root file system.
2890206   //
2890207   inode = NULL;
2890208   sb = sb_mount (DEV_DM02, &inode, MOUNT_DEFAULT);
2890209   if (sb == NULL || inode == NULL)
2890210     {
2890211       k_perror
2890212         ("Kernel panic: cannot mount " "root file system:");
2890213       k_exit ();
2890214     }
2890215   //
2890216   // Add the inode to the process table item for the
2890217   // kernel.
2890218   //
2890219   proc_table[0].inode_cwd = inode;      // Root fs
2890220   // inode.
2890221 }
```

94.14.4 kernel/proc/proc_print.c

«

Si veda la sezione 93.20.4.

```
2900001 #include <sys/os32.h>
2900002 #include <kernel/proc.h>
2900003 #include <kernel/lib_k.h>
2900004 //-----------------------------------------------------------
2900005 void
2900006 proc_print (void)
2900007 {
2900008   pid_t pid;
2900009   char stat;
2900010   unsigned int min;
2900011   unsigned int sec;
2900012   //
2900013   //
2900014   //
2900015   k_printf
2900016     ("pp  p pg                              T * 0x1000 "
2900017      "D * 0x1000 stack        \n"
2900018      "id id rp  tty  uid euid suid usage s addr  size "
2900019      "addr  size pointer  name\n");
2900020   //
2900021   //
```



```
2900022   //
2900023   for (pid = 0; pid < PROCESS_MAX; pid++)
2900024     {
2900025       if (proc_table[pid].status > 0)
2900026         {
2900027           switch (proc_table[pid].status)
2900028             {
2900029             case PROC_EMPTY:
2900030               stat = '-';
2900031               break;
2900032             case PROC_CREATED:
2900033               stat = 'c';
2900034               break;
2900035             case PROC_READY:
2900036               stat = 'r';
2900037               break;
2900038             case PROC_RUNNING:
2900039               stat = 'R';
2900040               break;
2900041             case PROC_SLEEPING:
2900042               stat = 's';
2900043               break;
2900044             case PROC_ZOMBIE:
2900045               stat = 'z';
2900046               break;
2900047             default:
2900048               stat = '?';
2900049               break;
2900050             }
2900051           //
2900052           min =
2900053             ((proc_table[pid].usage / CLOCKS_PER_SEC) / 60);
2900054           sec =
2900055             ((proc_table[pid].usage / CLOCKS_PER_SEC) % 60);
2900056           //
2900057           // Addresses and sizes are multiple of 4096
2900058           // (0x1000);
2900059           // for the stack pointer is shown only the
2900060           // last five
2900061           // hexadecimal digits.
2900062           //
2900063           if (proc_table[pid].domain_data > 0)
2900064             {
2900065               k_printf
2900066                 ("%2i %2i %2i %04x %4i %4i "
2900067                  "%4i %02i.%02i %c "
2900068                  "%05x %04x %05x %04x %07x %12s\n",
2900069                  (unsigned int) proc_table[pid].ppid,
2900070                  (unsigned int) pid,
2900071                  (unsigned int) proc_table[pid].pgrp,
2900072                  (unsigned int) proc_table[pid].device_tty,
2900073                  (unsigned int) proc_table[pid].uid,
2900074                  (unsigned int) proc_table[pid].euid,
2900075                  (unsigned int) proc_table[pid].suid,
2900076                  min, sec, stat,
2900077                  (unsigned int) proc_table[pid].address_text
2900078                  / MEM_BLOCK_SIZE,
2900079                  (unsigned int) proc_table[pid].domain_text
2900080                  / MEM_BLOCK_SIZE,
2900081                  (unsigned int) proc_table[pid].address_data
2900082                  / MEM_BLOCK_SIZE,
2900083                  (unsigned int) (proc_table[pid].domain_data
2900084                                  +
2900085                                  proc_table[pid].extra_data)
2900086                  / MEM_BLOCK_SIZE,
2900087                  (unsigned int) proc_table[pid].sp,
2900088                  proc_table[pid].name);
2900089             }
2900090           else
2900091             {
2900092               k_printf
2900093                 ("%2i %2i %2i %04x %4i %4i %4i "
2900094                  "%02i.%02i %c "
2900095                  "%05x %04x %05x %04x %07x %12s\n",
2900096                  (unsigned int) proc_table[pid].ppid,
2900097                  (unsigned int) pid,
2900098                  (unsigned int) proc_table[pid].pgrp,
2900099                  (unsigned int) proc_table[pid].device_tty,
2900100                  (unsigned int) proc_table[pid].uid,
2900101                  (unsigned int) proc_table[pid].euid,
2900102                  (unsigned int) proc_table[pid].suid,
2900103                  min, sec, stat,
2900104                  (unsigned int) proc_table[pid].address_text
2900105                  / MEM_BLOCK_SIZE,
2900106                  (unsigned int) (proc_table[pid].domain_text
2900107                                  +
2900108                                  proc_table[pid].extra_data)
```

```
                    / MEM_BLOCK_SIZE,
                    (unsigned int) proc_table[pid].address_data
                    / MEM_BLOCK_SIZE,
                    (unsigned int) proc_table[pid].domain_data
                    / MEM_BLOCK_SIZE,
                    (unsigned int) proc_table[pid].sp,
                    proc_table[pid].name);
                }
            }
        }
}
```

94.14.5 kernel/proc/proc_public.c

Si veda la sezione 93.20.5.

```
#include <kernel/proc.h>
//-------------------------------------------------------------
proc_t proc_table[PROCESS_MAX];
pid_t proc_current = 0;
uint16_t proc_stack_segment_selector = 24;
uint32_t proc_stack_pointer;
unsigned int proc_loops_per_clock;
```

94.14.6 kernel/proc/proc_reference.c

Si veda la sezione 93.20.5.

```
#include <kernel/proc.h>
//-------------------------------------------------------------
proc_t *
proc_reference (pid_t pid)
{
  if (pid >= 0 && pid < PROCESS_MAX)
    {
      return (&proc_table[pid]);
    }
  else
    {
      return (NULL);
    }
}
```

94.14.7 kernel/proc/proc_sch_net.c

Si veda la sezione 93.20.6.

```
#include <kernel/proc.h>
#include <kernel/lib_k.h>
#include <kernel/lib_s.h>
#include <kernel/driver/nic/ne2k.h>
#include <kernel/net.h>
#include <kernel/driver/pci.h>
//-------------------------------------------------------------
#define DEBUG 0
//-------------------------------------------------------------
void
proc_sch_net (void)
{
  int n;        // NET table index.
  int counter = 0;
  int pid;
  int tcp_wake_up;
  //
  // Do what there is to do with network interfaces,
  // in particular get
  // received frames, into the
  // 'net_table[n]...buffer[f]' table.
  //
  for (n = 0; n < NET_MAX_DEVICES; n++)
    {
      if (net_table[n].type == NET_DEV_ETH_NE2K)
        {
          //
          // The function 'ne2k_isr()' will call also
          // 'ne2k_rx()'
          // that is responsible for placing Ethernet
          // frames inside
          // 'ethernet_table[eth].frame_info[f]'.
          //
          ne2k_isr (net_table[n].ethernet.base_io);
        }
      else if (net_table[n].type == NET_DEV_LOOPBACK)
        {
          //
          // Packets should be already inside the
          // packet buffer table.
          //
          ;
        }
    }
  //
  // Do something with received Ethernet frames.
  //
  counter = net_rx ();
  //
  if (DEBUG)
    {
      if (counter)
        {
          k_printf ("+ %i packet(s) received\n", counter);
        }
    }
  //
  // Do something with TCP connections.
  //
  tcp_wake_up = tcp ();
  //
  // Wake up sleeping processes.
  //
  if (counter || tcp_wake_up == TCP_TRY_READ)
    {
      //
      // Wake up sleeping processes, waiting to read
      // from a socket.
      //
      if (DEBUG)
        {
          k_printf
            ("wake up processes, waiting "
             "for a socket!\n");
        }
      for (pid = 1; pid < PROCESS_MAX; pid++)
        {
          if (proc_table[pid].status == PROC_SLEEPING
              && (proc_table[pid].wakeup_events
                  & WAKEUP_EVENT_SOCK_READ))
            {
              proc_table[pid].wakeup_events = 0;
              proc_table[pid].status = PROC_READY;
            }
        }
    }
  //
  if (tcp_wake_up == TCP_TRY_WRITE)
    {
      //
      // Wake up sleeping processes, waiting to write
      // to a socket.
      //
      for (pid = 1; pid < PROCESS_MAX; pid++)
        {
          if (proc_table[pid].status == PROC_SLEEPING
              && (proc_table[pid].wakeup_events
                  & WAKEUP_EVENT_SOCK_WRITE))
            {
              proc_table[pid].wakeup_events = 0;
              proc_table[pid].status = PROC_READY;
            }
        }
    }
}
```

94.14.8 kernel/proc/proc_sch_signals.c

Si veda la sezione 93.20.7.

```
#include <kernel/proc.h>
//-------------------------------------------------------------
void
proc_sch_signals (void)
{
  pid_t pid;
  //
  // The PID scan starts from 1: you will not send
  // signals to the
  // kernel!
  //
  for (pid = 1; pid < PROCESS_MAX; pid++)
    {
      proc_sig_term (pid, SIGHUP);
      proc_sig_term (pid, SIGINT);
      proc_sig_core (pid, SIGQUIT);
      proc_sig_core (pid, SIGILL);
```

```
2940018         proc_sig_core (pid, SIGABRT);
2940019         proc_sig_core (pid, SIGFPE);
2940020         proc_sig_term (pid, SIGKILL);
2940021         proc_sig_core (pid, SIGSEGV);
2940022         proc_sig_term (pid, SIGPIPE);
2940023         proc_sig_term (pid, SIGALRM);
2940024         proc_sig_term (pid, SIGTERM);
2940025         proc_sig_term (pid, SIGUSR1);
2940026         proc_sig_term (pid, SIGUSR2);
2940027         proc_sig_chld (pid, SIGCHLD);
2940028         proc_sig_cont (pid, SIGCONT);
2940029         proc_sig_stop (pid, SIGSTOP);
2940030         proc_sig_stop (pid, SIGTSTP);
2940031         proc_sig_stop (pid, SIGTTIN);
2940032         proc_sig_stop (pid, SIGTTOU);
2940033       }
2940034   }
```

94.14.9 kernel/proc/proc_sch_terminals.c

«

Si veda la sezione 93.20.8.

```
2950001  #include <kernel/proc.h>
2950002  #include <kernel/lib_k.h>
2950003  #include <kernel/lib_s.h>
2950004  #include <kernel/driver/kbd.h>
2950005  #include <termios.h>
2950006  #include <limits.h>
2950007  //-------------------------------------------------------
2950008  void
2950009  proc_sch_terminals (void)
2950010  {
2950011    pid_t pid;
2950012    pid_t pid_sub;
2950013    unsigned char key;
2950014    tty_t *tty;
2950015    dev_t device;
2950016    int k;        // Reverse count killed character.
2950017    int overflow = 0;     // True if input is too much.
2950018    //
2950019    // Try to read a key from console keyboard buffer
2950020    // (only consoles
2950021    // are available). Variable 'kbd' is external and
2950022    // comes from
2950023    // 'kernel/kbd.h'.
2950024    //
2950025    key = kbd.key;
2950026    if (key == 0)
2950027      {
2950028        //
2950029        // No key is ready on the keyboard buffer: just
2950030        // return.
2950031        //
2950032        return;
2950033      }
2950034    else
2950035      {
2950036        //
2950037        // A key was pressed and it will be processed.
2950038        // Currently, just remove from the keyboard
2950039        // buffer.
2950040        kbd.key = 0;
2950041      }
2950042    //
2950043    // A key is available. Find the currently active
2950044    // console.
2950045    //
2950046    device = tty_console ((dev_t) 0);
2950047    tty = tty_reference (device);
2950048    if (tty == NULL)
2950049      {
2950050        k_printf
2950051          ("kernel alert: console device "
2950052           "0x%04x not found!\n", device);
2950053        //
2950054        // Will send the typed character to the first
2950055        // terminal!
2950056        //
2950057        tty = tty_reference ((dev_t) 0);
2950058      }
2950059    //
2950060    // Verify if it is a control key that must be
2950061    // handled before
2950062    // entering the canonical input line.
2950063    //
2950064    // Check for a console switch key combination.
2950065    //
```



```
2950066    if (key == 0x11)      // [Ctrl Q] -> DC1 ->
2950067      // console0.
2950068      {
2950069        tty_console (DEV_CONSOLE0);       // Switch.
2950070        return;
2950071      }
2950072    else if (key == 0x12) // [Ctrl R] -> DC2 ->
2950073      // console1.
2950074      {
2950075        tty_console (DEV_CONSOLE1);       // Switch.
2950076        return;
2950077      }
2950078    else if (key == 0x13) // [Ctrl S] -> DC3 ->
2950079      // console2.
2950080      {
2950081        tty_console (DEV_CONSOLE2);       // Switch.
2950082        return;
2950083      }
2950084    else if (key == 0x14) // [Ctrl T] -> DC4 ->
2950085      // console3.
2950086      {
2950087        tty_console (DEV_CONSOLE3);       // Switch.
2950088        return;
2950089      }
2950090    // -----------------------------------------------
2950091    // Only canonical input line is available: ICANON!
2950092    // -----------------------------------------------
2950093    //
2950094    // Might be necessary to send a signal to all
2950095    // processes with the
2950096    // same control terminal, excluded the kernel (0)
2950097    // and 'init' (1).
2950098    // Such control keys are not passed to the
2950099    // applications, or are
2950100    // replaced with zero.
2950101    //
2950102    // Please note that this a simplified solution,
2950103    // because the signal
2950104    // should reach only the foreground process of the
2950105    // group. For that
2950106    // reason, only che [Ctrl C] is taken into
2950107    // consideration, because
2950108    // processes can ignore the signal 'SIGINT'.
2950109    //
2950110    if (tty->pgrp != 0)
2950111      {
2950112        //
2950113        // There is a process group for that terminal.
2950114        //
2950115        if (tty->attr.c_cc[VINTR]
2950116            && key == tty->attr.c_cc[VINTR])
2950117          {
2950118            if (tty->attr.c_iflag & IGNBRK)
2950119              {
2950120                //
2950121                // No break!
2950122                //
2950123                return;
2950124              }
2950125            //
2950126            if (tty->attr.c_iflag & BRKINT)
2950127              {
2950128                if (tty->attr.c_lflag & ISIG)
2950129                  {
2950130                    for (pid = 2; pid < PROCESS_MAX; pid++)
2950131                      {
2950132                        if (proc_table[pid].pgrp == tty->pgrp)
2950133                          {
2950134                            //
2950135                            // Should find only final
2950136                            // process/processes.
2950137                            // So, if there is a son
2950138                            // process with the
2950139                            // same process group, the
2950140                            // signal is not
2950141                            // sent!
2950142                            //
2950143                            for (pid_sub = 2;
2950144                                 pid_sub < PROCESS_MAX;
2950145                                 pid_sub++)
2950146                              {
2950147                                if ((proc_table[pid_sub].ppid
2950148                                     == pid)
2950149                                    &&
2950150                                    (proc_table[pid_sub].pgrp
2950151                                     == tty->pgrp))
2950152                                  {
```

```
                                //
                                // 'pid_sub' is a
                                // son of
                                // 'pid' and is part
                                // of the
                                // same process
                                // group. 'pid_sub'
                                // is candidate for
                                // kill.
                                //
                                break;
                              }
                          }
                        //
                        if (pid_sub >= PROCESS_MAX)
                          {
                            //
                            // There is no son for
                            // 'pid', sorry.
                            //
                            s_kill ((pid_t) 0, pid,
                                    SIGINT);
                            //
                            // No more scan.
                            //
                            break;
                          }
                      }
                  }
              }
            //
            // Just reset input line and return.
            //
            tty->status = TTY_INPUT_LINE_EDITING;
            tty->lpr = 0;
            tty->lpw = 0;
            tty->line[0] = 0;
            //
            return;
          }
        //
        // Replace the INTR character with zero.
        //
        key = 0;
      }
  }
//
// Check if something is to be ignored.
//
if (key == '\r' && (tty->attr.c_iflag & IGNCR))
  {
    return;
  }
//
// Check if something is to be replaced, before
// editing.
//
if (key == '\n' && (tty->attr.c_iflag & INLCR))
  {
    key = '\r';
  }
//
if (key == '\r' && (tty->attr.c_iflag & ICRNL))
  {
    key = '\n';
  }
//
// Edit the canonical input line.
// Input is accepted only if status is ok.
//
if (tty->status == TTY_INPUT_LINE_EDITING)
  {
    //
    // Fix internal line positions.
    //
    if (tty->lpw < 0)
      {
        tty->lpw = 0;
      }
    if (tty->lpw >= MAX_CANON)
      {
        tty->lpw = MAX_CANON - 1;
        overflow = 1; // Too much input!
      }
    if (tty->lpr < 0)
      {
        tty->lpr = 0;
      }
    if (tty->lpr > tty->lpw)
      {
        tty->lpr = tty->lpw;
      }
    //
    // Check input key.
    //
    if (key == '\0')
      {
        //
        // A INTR replaced into zero.
        //
        tty->line[tty->lpw] = key;
        tty->status = TTY_INPUT_LINE_CLOSED;
        proc_wakeup_terminal ();
      }
    else if (key == '\n')
      {
        tty->line[tty->lpw] = key;
        tty->status = TTY_INPUT_LINE_CLOSED;
        proc_wakeup_terminal ();
      }
    else if (tty->attr.c_cc[VEOF]
             && key == tty->attr.c_cc[VEOF])
      {
        //
        // EOF is not included inside the line: just
        // replace the
        // with zero.
        //
        key = 0;
        tty->line[tty->lpw] = key;
        tty->status = TTY_INPUT_LINE_CLOSED;
        proc_wakeup_terminal ();
      }
    else if (tty->attr.c_cc[VEOL]
             && key == tty->attr.c_cc[VEOL])
      {
        tty->line[tty->lpw] = key;
        tty->status = TTY_INPUT_LINE_CLOSED;
        proc_wakeup_terminal ();
      }
    else if (tty->attr.c_cc[VERASE]
             && key == tty->attr.c_cc[VERASE])
      {
        //
        // Save how many characters to be killed, if
        // echo is
        // enabled.
        //
        if (overflow)
          {
            k = tty->lpw + 1;
            //
            // The 'tty->lpw' was already reduced.
            //
          }
        else
          {
            k = tty->lpw;
            //
            // Reduce write index: if it is less
            // than zero, it will
            // be fixed.
            //
            tty->lpw--;
          }
      }
    else if (tty->attr.c_cc[VKILL]
             && key == tty->attr.c_cc[VKILL])
      {
        //
        // Save how many characters to be killed, if
        // echo is
        // enabled.
        //
        if (overflow)
          {
            k = tty->lpw + 1;
          }
        else
          {
            k = tty->lpw;
          }
        //
        // Reset input line.
```

```
2950327           //
2950328           tty->lpw = 0;
2950329           tty->lpr = 0;
2950330           tty->line[0] = 0;
2950331         }
2950332       else
2950333         {
2950334           tty->line[tty->lpw] = key;
2950335           tty->lpw++;
2950336         }
2950337       //
2950338       // Echo.
2950339       //
2950340       if (!(tty->attr.c_lflag & ECHO))
2950341         {
2950342           //
2950343           // No echo. But NL might be echoed anyway.
2950344           //
2950345           if (key == '\n' && (tty->attr.c_lflag & ECHONL))
2950346             {
2950347               dev_io ((pid_t) 0, tty->device,
2950348                       DEV_WRITE, (off_t) 0, &key,
2950349                       (size_t) 1, NULL);
2950350             }
2950351           //
2950352           return;
2950353         }
2950354       //
2950355       // The echo is requested.
2950356       //
2950357       if (key == 0)
2950358         {
2950359           //
2950360           // There is nothing to echo.
2950361           //
2950362           ;
2950363         }
2950364       else if (tty->attr.c_cc[VERASE]
2950365                && key == tty->attr.c_cc[VERASE])
2950366         {
2950367           if (tty->attr.c_lflag & ECHOE)
2950368             {
2950369               if (k > 0)
2950370                 {
2950371                   dev_io ((pid_t) 0, tty->device,
2950372                           DEV_WRITE, (off_t) 0,
2950373                           "\b \b", (size_t) 3, NULL);
2950374                 }
2950375             }
2950376           else
2950377             {
2950378               dev_io ((pid_t) 0, tty->device,
2950379                       DEV_WRITE, (off_t) 0, &key,
2950380                       (size_t) 1, NULL);
2950381             }
2950382         }
2950383       else if (tty->attr.c_cc[VKILL]
2950384                && key == tty->attr.c_cc[VKILL])
2950385         {
2950386           if (tty->attr.c_lflag & ECHOK)
2950387             {
2950388               for (; k > 0; k--)
2950389                 {
2950390                   dev_io ((pid_t) 0, tty->device,
2950391                           DEV_WRITE, (off_t) 0,
2950392                           "\b \b", (size_t) 3, NULL);
2950393                 }
2950394             }
2950395         }
2950396       else if (key == '\n')
2950397         {
2950398           if (tty->attr.c_lflag & ECHONL)
2950399             {
2950400               dev_io ((pid_t) 0, tty->device,
2950401                       DEV_WRITE, (off_t) 0, &key,
2950402                       (size_t) 1, NULL);
2950403             }
2950404         }
2950405       else
2950406         {
2950407           //
2950408           // If there was an overflow, the last
2950409           // character is
2950410           // overwriting the last position, so a back
2950411           // space
2950412           // is printed.
2950413           //
2950414           if (overflow)
2950415             {
2950416               dev_io ((pid_t) 0, tty->device,
2950417                       DEV_WRITE, (off_t) 0, "\b",
2950418                       (size_t) 1, NULL);
2950419             }
2950420           //
2950421           // Now show the input character.
2950422           //
2950423           dev_io ((pid_t) 0, tty->device, DEV_WRITE,
2950424                   (off_t) 0, &key, (size_t) 1, NULL);
2950425         }
2950426     }
2950427 }
```

94.14.10 kernel/proc/proc_sch_timers.c

Si veda la sezione 93.20.9.

```
2960001 #include <kernel/proc.h>
2960002 #include <kernel/lib_k.h>
2960003 #include <kernel/lib_s.h>
2960004 //-------------------------------------------------------
2960005 void
2960006 proc_sch_timers (void)
2960007 {
2960008   static unsigned long long int previous_time;
2960009   unsigned long long int current_time;
2960010   unsigned int pid;
2960011   current_time = s_time ((pid_t) 0, NULL);
2960012   if (previous_time != current_time)
2960013     {
2960014       for (pid = 0; pid < PROCESS_MAX; pid++)
2960015         {
2960016           if ((proc_table[pid].wakeup_events &
2960017                WAKEUP_EVENT_TIMER)
2960018               && (proc_table[pid].status == PROC_SLEEPING)
2960019               && (proc_table[pid].wakeup_timer > 0))
2960020             {
2960021               proc_table[pid].wakeup_timer--;
2960022               if (proc_table[pid].wakeup_timer == 0)
2960023                 {
2960024                   proc_table[pid].status = PROC_READY;
2960025                 }
2960026             }
2960027         }
2960028     }
2960029   previous_time = current_time;
2960030 }
```

94.14.11 kernel/proc/proc_scheduler.c

Si veda la sezione 93.20.10.

```
2970001 #include <kernel/proc.h>
2970002 #include <kernel/lib_k.h>
2970003 #include <kernel/lib_s.h>
2970004 #include <kernel/net.h>
2970005 #include <stdint.h>
2970006 //-------------------------------------------------------
2970007 #define DEBUG 1
2970008 //-------------------------------------------------------
2970009 extern uint32_t _ksp;
2970010 extern uint32_t proc_stack_pointer;
2970011 extern uint16_t proc_stack_segment_selector;
2970012 extern pid_t proc_current;
2970013 //-------------------------------------------------------
2970014 void
2970015 proc_scheduler (void)
2970016 {
2970017   pid_t prev;
2970018   pid_t next;
2970019   addr_t stack_top;
2970020   addr_t stack_bottom;
2970021   uint32_t saved_stack_pointer = proc_stack_pointer;
2970022   //
2970023   static unsigned long long int previous_clock;
2970024   unsigned long long int current_clock;
2970025   //
2970026   // Check the current stack size.
2970027   //
2970028   if (proc_table[proc_current].domain_data == 0)
2970029     {
2970030       stack_bottom = proc_table[proc_current].domain_text;
2970031     }
2970032   else
2970033     {
```

```
2970034       stack_bottom = proc_table[proc_current].domain_data;
2970035     }
2970036   //
2970037   stack_top =
2970038     stack_bottom - proc_table[proc_current].domain_stack;
2970039   //
2970040   // Check if the process has broken data with the
2970041   // stack,
2970042   // or if it is near the end of its domain.
2970043   //
2970044   if (proc_stack_pointer <= stack_top)
2970045     {
2970046       //
2970047       // The stack overlaped the other data!
2970048       //
2970049       k_printf
2970050         ("[%s] Kernel alert: the stack of process %i "
2970051          "is grown beyond the allowed space! "
2970052          "The process "
2970053          "is closed. Stack top is %i, "
2970054          "stack pointer is %i.\n",
2970055          __FILE__, (int) proc_current, (int) stack_top,
2970056          (int) proc_stack_pointer);
2970057       //
2970058       // The process is terminated badly.
2970059       //
2970060       s__exit (proc_current, -1);
2970061     }
2970062   else if (proc_stack_pointer < (stack_top + 1024))
2970063     {
2970064       //
2970065       // There is only 1 Kibyte and the stack is
2970066       // finished!
2970067       //
2970068       k_printf
2970069         ("[%s] Kernel alert: the stack of process %i "
2970070          "is near the end of the allowed space! "
2970071          "It remains only %i byte and it will "
2970072          "overwrite other data!\n", __FILE__,
2970073          (int) proc_current,
2970074          (int) (proc_stack_pointer - stack_top));
2970075     }
2970076   //
2970077   // Save previous PID. Variable 'proc_current' is
2970078   // extern.
2970079   //
2970080   prev = proc_current;
2970081   //
2970082   // Take care of networking.
2970083   //
2970084   proc_sch_net ();
2970085   //
2970086   // Take care of sleeping processes: wake up if
2970087   // sleeping time
2970088   // elapsed.
2970089   //
2970090   proc_sch_timers ();
2970091   //
2970092   // Take care of pending signals.
2970093   //
2970094   proc_sch_signals ();
2970095   //
2970096   // Take care input from terminals.
2970097   //
2970098   proc_sch_terminals ();
2970099   //
2970100   // Update the CPU time usage.
2970101   //
2970102   current_clock = s_clock ((pid_t) 0);
2970103   proc_table[prev].usage += current_clock - previous_clock;
2970104   previous_clock = current_clock;
2970105   //
2970106   // Check stack pointer changes, made probably by
2970107   // 'proc_sig_handler()' called from
2970108   // 'proc_sch_signals()'.
2970109   //
2970110   if (DEBUG)
2970111     {
2970112       if (saved_stack_pointer != proc_stack_pointer)
2970113         {
2970114           k_printf
2970115             ("[%s] pid %i, ESP from %i to %i.\n",
2970116              __FILE__, proc_current,
2970117              saved_stack_pointer, proc_stack_pointer);
2970118         }
2970119     }
2970120   //
```

```
2970121   // Scan for a next process.
2970122   //
2970123   for (next = prev + 1; next != prev; next++)
2970124     {
2970125       if (next >= PROCESS_MAX)
2970126         {
2970127           next = -1;    // At the next loop, 'next'
2970128           // will be zero.
2970129           continue;
2970130         }
2970131       //
2970132       if (proc_table[next].status == PROC_EMPTY)
2970133         {
2970134           continue;
2970135         }
2970136       else if (proc_table[next].status == PROC_CREATED)
2970137         {
2970138           continue;
2970139         }
2970140       else if (proc_table[next].status == PROC_READY)
2970141         {
2970142           if (proc_table[prev].status == PROC_RUNNING)
2970143             {
2970144               proc_table[prev].status = PROC_READY;
2970145             }
2970146           //
2970147           proc_table[prev].sp = proc_stack_pointer;
2970148           proc_table[next].status = PROC_RUNNING;
2970149           proc_table[next].ret = 0;
2970150           //
2970151           proc_current = next;
2970152           proc_stack_segment_selector
2970153             = gdt_pid_to_segment_data (next) * 8;
2970154           proc_stack_pointer = proc_table[next].sp;
2970155           break;
2970156         }
2970157       else if (proc_table[next].status == PROC_RUNNING)
2970158         {
2970159           if (proc_table[prev].status == PROC_RUNNING)
2970160             {
2970161               k_printf ("Kernel alert: process %i "
2970162                         "and %i \"running\"!\n",
2970163                         prev, next);
2970164               proc_table[prev].status = PROC_READY;
2970165             }
2970166           //
2970167           proc_table[prev].sp = proc_stack_pointer;
2970168           proc_table[next].status = PROC_RUNNING;
2970169           proc_table[next].ret = 0;
2970170           //
2970171           proc_current = next;
2970172           proc_stack_segment_selector
2970173             = gdt_pid_to_segment_data (next) * 8;
2970174           proc_stack_pointer = proc_table[next].sp;
2970175           break;
2970176         }
2970177       else if (proc_table[next].status == PROC_SLEEPING)
2970178         {
2970179           continue;
2970180         }
2970181       else if (proc_table[next].status == PROC_ZOMBIE)
2970182         {
2970183           continue;
2970184         }
2970185     }
2970186   //
2970187   // Check again if the next process is set to
2970188   // running, otherwise set
2970189   // the kernel to such value!
2970190   //
2970191   if (proc_table[next].status != PROC_RUNNING)
2970192     {
2970193       proc_table[0].status = PROC_RUNNING;
2970194       proc_current = 0;
2970195       proc_stack_segment_selector
2970196         = gdt_pid_to_segment_data (0) * 8;
2970197       proc_stack_pointer = proc_table[0].sp;
2970198     }
2970199   //
2970200   // Save kernel stack pointer.
2970201   //
2970202   _ksp = proc_table[0].sp;
2970203 }
```

94.14.12 kernel/proc/proc_sig_chld.c

Si veda la sezione 93.20.11.

```
2980001 #include <kernel/proc.h>
2980002 //---------------------------------------------------------
2980003 // At the moment, the SIGCHLD is handled only per
2980004 // default: no other handler is taken into
2980005 // consideration.
2980006 //---------------------------------------------------------
2980007 void
2980008 proc_sig_chld (pid_t parent, int sig)
2980009 {
2980010   pid_t child;
2980011   //
2980012   // Please note that 'sig' should be SIGCHLD and
2980013   // nothing else.
2980014   // So, the following test, means to verify if the
2980015   // parent process
2980016   // has received a SIGCHLD already.
2980017   //
2980018   if (proc_sig_status (parent, sig))
2980019     {
2980020       if ((!proc_sig_ignore (parent, sig))
2980021           && (proc_table[parent].status ==
2980022               PROC_SLEEPING)
2980023           && (proc_table[parent].wakeup_events &
2980024               WAKEUP_EVENT_SIGNAL)
2980025           && (proc_table[parent].wakeup_signal == sig))
2980026         {
2980027           //
2980028           // The signal is not ignored from the parent
2980029           // process;
2980030           // the parent process is sleeping;
2980031           // the parent process is waiting for a
2980032           // signal;
2980033           // the parent process is waiting for current
2980034           // signal.
2980035           // So, just wake it up.
2980036           //
2980037           proc_table[parent].status = PROC_READY;
2980038           proc_table[parent].wakeup_events = 0;
2980039           proc_table[parent].wakeup_signal = 0;
2980040         }
2980041       else
2980042         {
2980043           //
2980044           // All other cases, means to remove all dead
2980045           // children.
2980046           //
2980047           for (child = 1; child < PROCESS_MAX; child++)
2980048             {
2980049               if (proc_table[child].ppid == parent
2980050                   && proc_table[child].status ==
2980051                   PROC_ZOMBIE)
2980052                 {
2980053                   proc_available (child);
2980054                 }
2980055             }
2980056         }
2980057       proc_sig_off (parent, sig);
2980058     }
2980059 }
```

94.14.13 kernel/proc/proc_sig_cont.c

Si veda la sezione 93.20.12.

```
2990001 #include <kernel/proc.h>
2990002 //---------------------------------------------------------
2990003 void
2990004 proc_sig_cont (pid_t pid, int sig)
2990005 {
2990006   //
2990007   // The value for argument 'sig' should be SIGCONT.
2990008   //
2990009   if (proc_sig_status (pid, sig))
2990010     {
2990011       if (proc_sig_ignore (pid, sig))
2990012         {
2990013           proc_sig_off (pid, sig);
2990014         }
2990015       else
2990016         {
2990017           if (proc_table[pid].sig_handler[sig] !=
2990018               (uintptr_t) NULL)
2990019             {
2990020               proc_sig_handler (pid, sig);
2990021             }
2990022           else
2990023             {
2990024               proc_table[pid].status = PROC_READY;
2990025               proc_sig_off (pid, sig);
2990026             }
2990027         }
2990028     }
2990029 }
```


94.14.14 kernel/proc/proc_sig_core.c

Si veda la sezione 93.20.13.

```
3000001 #include <kernel/proc.h>
3000002 #include <kernel/lib_s.h>
3000003 //---------------------------------------------------------
3000004 void
3000005 proc_sig_core (pid_t pid, int sig)
3000006 {
3000007   addr_t address_text;
3000008   addr_t address_data;
3000009   size_t domain_text;
3000010   size_t domain_data;
3000011   size_t extra_data;
3000012   //
3000013   if (proc_sig_status (pid, sig))
3000014     {
3000015       if (proc_sig_ignore (pid, sig))
3000016         {
3000017           proc_sig_off (pid, sig);
3000018         }
3000019       else
3000020         {
3000021           if (proc_table[pid].sig_handler[sig] !=
3000022               (uintptr_t) NULL)
3000023             {
3000024               proc_sig_handler (pid, sig);
3000025             }
3000026           else
3000027             {
3000028               //
3000029               // Save process addresses and sizes
3000030               // (might be useful if
3000031               // we want to try to exit the process
3000032               // before core dump.
3000033               //
3000034               address_text = proc_table[pid].address_text;
3000035               address_data = proc_table[pid].address_data;
3000036               domain_text = proc_table[pid].domain_text;
3000037               domain_data = proc_table[pid].domain_data;
3000038               extra_data = proc_table[pid].extra_data;
3000039               //
3000040               // Core dump: the process who formally
3000041               // writes the file
3000042               // is the terminating one.
3000043               //
3000044               if (domain_data == 0)
3000045                 {
3000046                   proc_dump_memory (pid, address_text,
3000047                                     domain_text +
3000048                                     extra_data, "core");
3000049                 }
3000050               else
3000051                 {
3000052                   proc_dump_memory (pid, address_text,
3000053                                     domain_text,
3000054                                     "core.text");
3000055                   proc_dump_memory (pid, address_data,
3000056                                     domain_data +
3000057                                     extra_data,
3000058                                     "core.data");
3000059                 }
3000060               //
3000061               // The signal, translated to negative,
3000062               // is returned (but
3000063               // the effective value received by the
3000064               // application will
3000065               // be cutted, leaving only the low 8
3000066               // bit).
3000067               //
3000068               s__exit (pid, -sig);
3000069             }
3000070         }
3000071     }
3000072 }
```

94.14.15 kernel/proc/proc_sig_handler.c

«

Si veda la sezione 93.20.14.

```
3010001 #include <kernel/proc.h>
3010002 #include <kernel/lib_s.h>
3010003 #include <kernel/lib_k.h>
3010004 #include <stdint.h>
3010005 //-----------------------------------------------------------
3010006 #define DEBUG 0
3010007 //-----------------------------------------------------------
3010008 void
3010009 proc_sig_handler (pid_t pid, int sig)
3010010 {
3010011   addr_t addr_data_top;
3010012   addr_t addr_stack_pointer;
3010013   uint32_t old_eip;
3010014   uint32_t old_cs;
3010015   uint32_t old_eflags;
3010016   //
3010017   // Stack frames.
3010018   //
3010019   struct
3010020   {
3010021     uint32_t eax;
3010022     uint32_t ecx;
3010023     uint32_t edx;
3010024     uint32_t ebx;
3010025     uint32_t ebp;
3010026     uint32_t esi;
3010027     uint32_t edi;
3010028     uint32_t ds;
3010029     uint32_t es;
3010030     uint32_t fs;
3010031     uint32_t gs;
3010032     uint32_t eip;
3010033     uint32_t cs;
3010034     uint32_t eflags;
3010035   } *old;
3010036   //
3010037   struct
3010038   {
3010039     uint32_t eax;
3010040     uint32_t ecx;
3010041     uint32_t edx;
3010042     uint32_t ebx;
3010043     uint32_t ebp;
3010044     uint32_t esi;
3010045     uint32_t edi;
3010046     uint32_t ds;
3010047     uint32_t es;
3010048     uint32_t fs;
3010049     uint32_t gs;
3010050     uint32_t wrapper;
3010051     uint32_t cs;
3010052     uint32_t eflags;
3010053     uint32_t handler;
3010054     uint32_t signal;
3010055     uint32_t eip;
3010056   } *new;
3010057   //
3010058   // First of all, this function can act only for a
3010059   // process that
3010060   // is not *just* interrupted. That is, if
3010061   // 'proc_current' is
3010062   // equal to 'pid', nothing is to be done yet: it
3010063   // will be done
3010064   // only when it will be in pause.
3010065   //
3010066   if (pid == proc_current)
3010067     {
3010068       //
3010069       // Nothing to do yet.
3010070       //
3010071       return;
3010072     }
3010073   //
3010074   // Check if there is a function to run.
3010075   //
3010076   if (proc_table[pid].sig_handler[sig] == (uintptr_t) NULL)
3010077     {
3010078       //
3010079       // Nothing to do.
3010080       //
3010081       return;
3010082     }
3010083   //
3010084   // Tell something for debugging.
3010085   //
3010086   if (DEBUG)
3010087     {
3010088       k_printf ("%s(%i, %i)", __func__, (int) pid, sig);
3010089     }
3010090   //
3010091   // Calculate the absolute stack section address,
3010092   // from the
3010093   // kernel point of view.
3010094   //
3010095   if (proc_table[pid].domain_data == 0)
3010096     {
3010097       addr_data_top = proc_table[pid].address_text;
3010098     }
3010099   else
3010100     {
3010101       addr_data_top = proc_table[pid].address_data;
3010102     }
3010103   //
3010104   // Then calculate the effective stack pointer
3010105   // address.
3010106   // We are considering only process that are not
3010107   // currently interrupted, so the stack pointer is
3010108   // taken from 'proc_table[pid].sp'.
3010109   //
3010110   addr_stack_pointer = addr_data_top + proc_table[pid].sp;
3010111   //
3010112   // Currently the process stack is as it follows.
3010113   // The address inside 'addr_stack_pointer' is
3010114   // corresponding
3010115   // to the saved EAX value.
3010116   //
3010117   // pushl %eflags #
3010118   // pushl %cs # from the interrupt
3010119   // pushl %eip #
3010120   // -------------
3010121   // pushl %gs
3010122   // pushl %fs
3010123   // pushl %es
3010124   // pushl %ds
3010125   // pushl %edi
3010126   // pushl %esi
3010127   // pushl %ebp
3010128   // pushl %ebx
3010129   // pushl %edx
3010130   // pushl %ecx
3010131   // pushl %eax
3010132   //
3010133   // Need to insert the call to the handler function.
3010134   // The process stack should become this way:
3010135   //
3010136   // pushl %eip [0]
3010137   // pushl <signal> [1]
3010138   // pushl <handler> [1]
3010139   // pushl %eflags [2]
3010140   // pushl %cs [2]
3010141   // pushl <wrapper> [2]
3010142   // -------------
3010143   // pushl %gs
3010144   // pushl %fs
3010145   // pushl %es
3010146   // pushl %ds
3010147   // pushl %edi
3010148   // pushl %esi
3010149   // pushl %ebp
3010150   // pushl %ebx
3010151   // pushl %edx
3010152   // pushl %ecx
3010153   // pushl %eax
3010154   //
3010155   // [0] Back from the original interrupt.
3010156   //
3010157   // [1] Arguments of the wrapper functions, that will
3010158   // call the
3010159   // signal handler, and then will return to the
3010160   // address at
3010161   // [1].
3010162   //
3010163   // [2] Modified so that the IRET instruction will
3010164   // return
3010165   // to the begin of the wrapper function, that will
3010166   // call the true signal handler, and will return
3010167   // at [1].
3010168   //
3010169   // Now set the pointer to the old and new frame,
3010170   // updating the stack pointer address.
3010171   //
3010172   old = (void *) addr_stack_pointer;
```

```
3010173   addr_stack_pointer -= 12;     // Three more
3010174   // elements.
3010175   new = (void *) addr_stack_pointer;
3010176   //
3010177   // Verify if the code segment was correctly found.
3010178   //
3010179   if (DEBUG)
3010180     {
3010181       k_printf
3010182         ("[%s] EAX:%04x ECX:%04x EDX:%04x "
3010183          "EBX:%04x EBP:%04x "
3010184          "ESI:%04x EDI:%04x DS:%04x ES:%04x "
3010185          "FS:%04x GS:%04x "
3010186          "EIP:%04x CS:%04x EFLAGS:%04x\n", __FILE__,
3010187          (int) old->eax, (int) old->ecx,
3010188          (int) old->edx, (int) old->ebx,
3010189          (int) old->ebp, (int) old->esi,
3010190          (int) old->edi, (int) old->ds, (int) old->es,
3010191          (int) old->fs, (int) old->gs, (int) old->eip,
3010192          (int) old->cs, (int) old->eflags);
3010193     }
3010194   //
3010195   // Move data, to arrange the new stack. The order
3010196   // does
3010197   // matter, as the new frame overwrites the old one.
3010198   //
3010199   new->eax = old->eax;
3010200   new->ecx = old->ecx;
3010201   new->edx = old->edx;
3010202   new->ebx = old->ebx;
3010203   new->ebp = old->ebp;
3010204   new->esi = old->esi;
3010205   new->edi = old->edi;
3010206   new->ds = old->ds;
3010207   new->es = old->es;
3010208   new->fs = old->fs;
3010209   new->gs = old->gs;
3010210   //
3010211   old_eflags = old->eflags;
3010212   old_cs = old->cs;
3010213   old_eip = old->eip;
3010214   //
3010215   new->wrapper = proc_table[pid].sig_handler_wrapper;
3010216   new->cs = old_cs;
3010217   new->eflags = old_eflags;
3010218   new->handler = proc_table[pid].sig_handler[sig];
3010219   new->signal = sig;
3010220   new->eip = old_eip;
3010221   //
3010222   // Tell what is changed inside the stack.
3010223   //
3010224   if (DEBUG)
3010225     {
3010226       k_printf
3010227         ("[%s] EAX:%04x ECX:%04x EDX:%04x "
3010228          "EBX:%04x EBP:%04x "
3010229          "ESI:%04x EDI:%04x DS:%04x ES:%04x "
3010230          "FS:%04x GS:%04x "
3010231          "wrapper:%04x CS:%04x EFLAGS:%04x "
3010232          "handler:%04x "
3010233          "signal:$04x EIP:%04x\n", __FILE__,
3010234          (int) new->eax, (int) new->ecx,
3010235          (int) new->edx, (int) new->ebx,
3010236          (int) new->ebp, (int) new->esi,
3010237          (int) new->edi, (int) new->ds, (int) new->es,
3010238          (int) new->fs, (int) new->gs,
3010239          (int) new->wrapper, (int) new->cs,
3010240          (int) new->eflags, (int) new->handler,
3010241          (int) new->signal, (int) new->eip);
3010242     }
3010243   //
3010244   // Change the stack pointer of the process, as it
3010245   // was increased.
3010246   //
3010247   proc_table[pid].sp = addr_stack_pointer - addr_data_top;
3010248   //
3010249   // Reset the signal handler, as in traditional Unix,
3010250   // with
3010251   // all the consequences that such implementation
3010252   // will give.
3010253   //
3010254   proc_table[pid].sig_handler[sig] = (uintptr_t) NULL;
3010255   proc_sig_off (pid, sig);
3010256   //
3010257   // Wake up the process if it is sleeping.
3010258   //
3010259   proc_table[pid].wakeup_events = 0;
3010260   proc_table[pid].status = PROC_READY;
3010261 }
```


94.14.16 kernel/proc/proc_sig_ignore.c

Si veda la sezione 93.20.15.

```
3020001 #include <kernel/proc.h>
3020002 //-------------------------------------------------------
3020003 int
3020004 proc_sig_ignore (pid_t pid, int sig)
3020005 {
3020006   unsigned long int flag = 1L << (sig - 1);
3020007   if (proc_table[pid].sig_ignore & flag)
3020008     {
3020009       return (1);
3020010     }
3020011   else
3020012     {
3020013       return (0);
3020014     }
3020015 }
```

94.14.17 kernel/proc/proc_sig_off.c

Si veda la sezione 93.20.17.

```
3030001 #include <kernel/proc.h>
3030002 //-------------------------------------------------------
3030003 void
3030004 proc_sig_off (pid_t pid, int sig)
3030005 {
3030006   unsigned long int flag = 1L << (sig - 1);
3030007   proc_table[pid].sig_status ^= flag;
3030008 }
```

94.14.18 kernel/proc/proc_sig_on.c

Si veda la sezione 93.20.17.

```
3040001 #include <kernel/proc.h>
3040002 //-------------------------------------------------------
3040003 void
3040004 proc_sig_on (pid_t pid, int sig)
3040005 {
3040006   unsigned long int flag = 1L << (sig - 1);
3040007   proc_table[pid].sig_status |= flag;
3040008 }
```

94.14.19 kernel/proc/proc_sig_status.c

Si veda la sezione 93.20.18.

```
3050001 #include <kernel/proc.h>
3050002 //-------------------------------------------------------
3050003 int
3050004 proc_sig_status (pid_t pid, int sig)
3050005 {
3050006   unsigned long int flag = 1L << (sig - 1);
3050007   if (proc_table[pid].sig_status & flag)
3050008     {
3050009       return (1);
3050010     }
3050011   else
3050012     {
3050013       return (0);
3050014     }
3050015 }
```

94.14.20 kernel/proc/proc_sig_stop.c

Si veda la sezione 93.20.19.

```
3060001 #include <kernel/proc.h>
3060002 //-------------------------------------------------------
3060003 void
3060004 proc_sig_stop (pid_t pid, int sig)
3060005 {
3060006   if (proc_sig_status (pid, sig))
3060007     {
3060008       if (proc_sig_ignore (pid, sig) && !(sig == SIGSTOP))
3060009         {
3060010           proc_sig_off (pid, sig);
3060011         }
3060012       else
3060013         {
```

```
3060014             if ((proc_table[pid].sig_handler[sig] !=
3060015                  (uintptr_t) NULL) && (sig != SIGSTOP))
3060016               {
3060017                 proc_sig_handler (pid, sig);
3060018               }
3060019             else
3060020               {
3060021                 proc_table[pid].status = PROC_SLEEPING;
3060022                 proc_table[pid].ret = -sig;
3060023                 proc_sig_off (pid, sig);
3060024               }
3060025           }
3060026       }
3060027 }
```

94.14.21 kernel/proc/proc_sig_term.c

«

Si veda la sezione 93.20.20.

```
3070001 #include <kernel/proc.h>
3070002 #include <kernel/lib_s.h>
3070003 #include <kernel/lib_k.h>
3070004 //-----------------------------------------------------------
3070005 void
3070006 proc_sig_term (pid_t pid, int sig)
3070007 {
3070008   if (proc_sig_status (pid, sig))
3070009     {
3070010       if (proc_sig_ignore (pid, sig) && !(sig == SIGKILL))
3070011         {
3070012           proc_sig_off (pid, sig);
3070013         }
3070014       else
3070015         {
3070016           if ((proc_table[pid].sig_handler[sig] !=
3070017                (uintptr_t) NULL) && (sig != SIGKILL))
3070018             {
3070019               proc_sig_handler (pid, sig);
3070020             }
3070021           else
3070022             {
3070023               //
3070024               // The signal, translated to negative,
3070025               // is returned (but
3070026               // the effective value received by the
3070027               // application will
3070028               // be cutted, leaving only the low 8
3070029               // bit).
3070030               //
3070031               s__exit (pid, -sig);
3070032             }
3070033         }
3070034     }
3070035 }
```

94.14.22 kernel/proc/proc_sys_exec.c

«

Si veda la sezione 93.20.21.

```
3080001 #include <kernel/ibm_i386.h>
3080002 #include <kernel/proc.h>
3080003 #include <errno.h>
3080004 #include <fcntl.h>
3080005 #include <kernel/lib_s.h>
3080006 #include <kernel/lib_k.h>
3080007 //-----------------------------------------------------------
3080008 #define DEBUG 0
3080009 //-----------------------------------------------------------
3080010 int
3080011 proc_sys_exec (pid_t pid, const char *path,
3080012                unsigned int argc, char *arg_data,
3080013                unsigned int envc, char *env_data)
3080014 {
3080015   unsigned int i;
3080016   unsigned int j;
3080017   char *arg;
3080018   char *env;
3080019   char *envp[ARG_MAX / 16];
3080020   char *argv[ARG_MAX / 16];
3080021   size_t size;
3080022   size_t arg_data_size;
3080023   size_t env_data_size;
3080024   unsigned int p_off;
3080025   inode_t *inode;
3080026   ssize_t size_read;
3080027   header_t header;
3080028   uint32_t new_sp;
```



```
3080029   uint32_t envp_address;
3080030   uint32_t argv_address;
3080031   addr_t allocated_text;
3080032   addr_t allocated_data;
3080033   addr_t stack_location;        // real stack
3080034   // location.
3080035   size_t process_domain_text;
3080036   size_t process_domain_data;
3080037   size_t process_domain_stack;
3080038   addr_t previous_address_text;
3080039   addr_t previous_address_data;
3080040   size_t previous_domain_text;
3080041   size_t previous_domain_data;
3080042   size_t previous_domain_stack;
3080043   size_t previous_extra_data;
3080044   uint32_t segment_text;        // Segment descriptors
3080045   uint32_t segment_data;        // inside 32 bit int.
3080046   char buffer[MEM_BLOCK_SIZE];
3080047   uint32_t stack_element;
3080048   off_t off_inode;
3080049   addr_t memory_start;
3080050   int status;
3080051   pid_t extra;
3080052   int proc_count;
3080053   file_t *file;
3080054   int fdn;
3080055   dev_t device;
3080056   int eof;
3080057   int sig;
3080058   //
3080059   // Check for limits.
3080060   //
3080061   if (argc > (ARG_MAX / 16) || envc > (ARG_MAX / 16))
3080062     {
3080063       errset (ENOMEM);
3080064       return (-1);
3080065     }
3080066   //
3080067   // Scan arguments to calculate the full size and the
3080068   // relative
3080069   // pointers. The final size is rounded to 4, for the
3080070   // stack.
3080071   //
3080072   arg = arg_data;
3080073   for (i = 0, j = 0; i < argc; i++)
3080074     {
3080075       argv[i] = (char *) j;     // Relative pointer
3080076       // inside the
3080077       // 'arg_data'.
3080078       size = strlen (arg);
3080079       arg += size + 1;
3080080       j += size + 1;
3080081     }
3080082   arg_data_size = j;
3080083   if (arg_data_size % 2)
3080084     {
3080085       arg_data_size++;
3080086     }
3080087   if (arg_data_size % 4)
3080088     {
3080089       arg_data_size += 2;
3080090     }
3080091   //
3080092   // Scan environment variables to calculate the full
3080093   // size and the
3080094   // relative pointers. The final size is rounded to
3080095   // 4, for the stack.
3080096   //
3080097   env = env_data;
3080098   for (i = 0, j = 0; i < envc; i++)
3080099     {
3080100       envp[i] = (char *) j;     // Relative pointer
3080101       // inside the
3080102       // 'env_data'.
3080103       size = strlen (env);
3080104       env += size + 1;
3080105       j += size + 1;
3080106     }
3080107   env_data_size = j;
3080108   if (env_data_size % 2)
3080109     {
3080110       env_data_size++;
3080111     }
3080112   if (env_data_size % 4)
3080113     {
3080114       env_data_size += 2;
3080115     }
```

```
3080116   //
3080117   // Read the inode related to the executable file
3080118   // name.
3080119   // Function path_inode() includes the inode get
3080120   // procedure.
3080121   //
3080122   inode = path_inode (pid, path);
3080123   if (inode == NULL)
3080124     {
3080125       errset (ENOENT);  // No such file or directory.
3080126       return (-1);
3080127     }
3080128   //
3080129   // Check for permissions.
3080130   //
3080131   status =
3080132     inode_check (inode, S_IFREG, 5,
3080133                  proc_table[pid].euid,
3080134                  proc_table[pid].egid);
3080135   if (status != 0)
3080136     {
3080137       //
3080138       // File is not of a valid type or permission are
3080139       // not
3080140       // sufficient: release the executable file inode
3080141       // and return with an error.
3080142       //
3080143       inode_put (inode);
3080144       errset (EACCES);  // Permission denied.
3080145       return (-1);
3080146     }
3080147   //
3080148   // Read the header from the executable file.
3080149   //
3080150   size_read =
3080151     inode_file_read (inode, (off_t) 0, &header,
3080152                      (sizeof header), &eof);
3080153   if (size_read != (sizeof header))
3080154     {
3080155       //
3080156       // The file is shorter than the executable
3080157       // header, so, it isn't
3080158       // an executable: release the file inode and
3080159       // return with an
3080160       // error.
3080161       //
3080162       inode_put (inode);
3080163       errset (ENOEXEC);
3080164       return (-1);
3080165     }
3080166   //
3080167   // Size read is ok.
3080168   //
3080169   if (header.magic != MAGIC_OS32_APPL)
3080170     {
3080171       //
3080172       // The header does not have the expected magic
3080173       // numbers, so,
3080174       // it isn't a valid executable: release the file
3080175       // inode and
3080176       // return with an error.
3080177       //
3080178       inode_put (inode);
3080179       errset (ENOEXEC);
3080180       return (-1);      // This is not a valid
3080181       // executable!
3080182     }
3080183   //
3080184   // Calculate code size.
3080185   //
3080186   if (header.data_offset == 0)
3080187     {
3080188       process_domain_text = header.ebss + header.ssize;
3080189     }
3080190   else
3080191     {
3080192       process_domain_text = header.data_offset;
3080193     }
3080194   //
3080195   if (process_domain_text % 4096)
3080196     {
3080197       process_domain_text =
3080198         (((process_domain_text / 4096) + 1) * 4096);
3080199     }
3080200   //
3080201   // Calculate data size, including stack, that cannot
3080202   // stay alone!
```

```
3080203   //
3080204   process_domain_stack = header.ssize;
3080205   //
3080206   if (header.data_offset == 0)
3080207     {
3080208       process_domain_data = 0;
3080209     }
3080210   else
3080211     {
3080212       process_domain_data = (header.ebss + header.ssize);
3080213     }
3080214   //
3080215   if (process_domain_data % 4096)
3080216     {
3080217       process_domain_data =
3080218         (((process_domain_data / 4096) + 1) * 4096);
3080219     }
3080220   //
3080221   // Place the new stack pointer to the bottom of the
3080222   // data area:
3080223   // the stack pointer is relative to the data area,
3080224   // so the last
3080225   // relative position is equal to the size.
3080226   //
3080227   if (header.data_offset == 0)
3080228     {
3080229       new_sp = process_domain_text;
3080230     }
3080231   else
3080232     {
3080233       new_sp = process_domain_data;
3080234     }
3080235   //
3080236   // Allocate memory: code and data.
3080237   //
3080238   allocated_text = mb_alloc_size (process_domain_text);
3080239   //
3080240   if (allocated_text == 0)
3080241     {
3080242       //
3080243       // The program instructions (code segment)
3080244       // cannot be loaded
3080245       // into memory: release the executable file
3080246       // inode and return
3080247       // with an error.
3080248       //
3080249       inode_put (inode);
3080250       errset (ENOMEM);  // Not enough space.
3080251       return (-1);
3080252     }
3080253   else if (DEBUG)
3080254     {
3080255       k_printf ("%s:%i:mb_alloc_size(%zi)", __FILE__,
3080256                 __LINE__,
3080257                 (unsigned int) process_domain_text);
3080258     }
3080259   //
3080260   //
3080261   //
3080262   if (header.data_offset == 0)
3080263     {
3080264       //
3080265       // Code and data segments are the same: no need
3080266       // to allocate more memory for the data segment.
3080267       //
3080268       allocated_data = 0;
3080269       process_domain_data = 0;
3080270     }
3080271   else
3080272     {
3080273       //
3080274       // Code and data segments are different: the
3080275       // data
3080276       // segment memory is allocated.
3080277       //
3080278       allocated_data = mb_alloc_size (process_domain_data);
3080279       if (allocated_data == 0)
3080280         {
3080281           //
3080282           // The separated program data (data segment)
3080283           // cannot be
3080284           // loaded into memory: free the already
3080285           // allocated memory
3080286           // for the program instructions, release the
3080287           // executable
3080288           // file inode and return with an error.
3080289           //
```

```
3080290           mb_free (allocated_text, process_domain_text);
3080291           if (DEBUG)
3080292             {
3080293               k_printf ("%s:%i:mb_free(%i, %zi)",
3080294                         __FILE__, __LINE__,
3080295                         (unsigned int) allocated_text,
3080296                         process_domain_data);
3080297             }
3080298           inode_put (inode);
3080299           errset (ENOMEM);      // Not enough space.
3080300           return (-1);
3080301         }
3080302       else if (DEBUG)
3080303         {
3080304           k_printf ("%s:%i:mb_alloc_size(%zi)",
3080305                     __FILE__, __LINE__,
3080306                     process_domain_data);
3080307         }
3080308     }
3080309   //
3080310   // Load executable in memory.
3080311   //
3080312   if (header.data_offset == 0)
3080313     {
3080314       //
3080315       // Code and data share the same segment.
3080316       //
3080317       for (eof = 0, memory_start = allocated_text,
3080318            off_inode = 0, size_read = 0;
3080319            off_inode < inode->size && !eof;
3080320            off_inode += size_read)
3080321         {
3080322           memory_start += size_read;
3080323           //
3080324           // Read a block of memory.
3080325           //
3080326           size_read =
3080327             inode_file_read (inode, off_inode, buffer,
3080328                              MEM_BLOCK_SIZE, &eof);
3080329           if (size_read < 0)
3080330             {
3080331               //
3080332               // Free memory and inode.
3080333               //
3080334               mb_free (allocated_text, process_domain_text);
3080335               if (DEBUG)
3080336                 {
3080337                   k_printf ("%s:%i:mb_free(%i, %zi)",
3080338                             __FILE__, __LINE__,
3080339                             (unsigned int)
3080340                             allocated_text,
3080341                             process_domain_text);
3080342                 }
3080343               inode_put (inode);
3080344               errset (EIO);
3080345               return (-1);
3080346             }
3080347           //
3080348           // Copy inside the right position to be
3080349           // executed.
3080350           //
3080351           dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080352                   memory_start, buffer,
3080353                   (size_t) size_read, NULL);
3080354         }
3080355     }
3080356   else
3080357     {
3080358       //
3080359       // Code and data with different segments.
3080360       //
3080361       for (eof = 0, memory_start = allocated_text,
3080362            off_inode = 0, size_read = 0;
3080363            off_inode < process_domain_text && !eof;
3080364            off_inode += size_read)
3080365         {
3080366           memory_start += size_read;
3080367           //
3080368           // Read a block of memory
3080369           //
3080370           size_read =
3080371             inode_file_read (inode, off_inode, buffer,
3080372                              MEM_BLOCK_SIZE, &eof);
3080373           if (size_read < 0)
3080374             {
3080375               //
3080376               // Free memory and inode.
```

```
3080377               //
3080378               mb_free (allocated_text, process_domain_text);
3080379               if (DEBUG)
3080380                 {
3080381                   k_printf ("%s:%i:mb_free(%i, %zi)",
3080382                             __FILE__, __LINE__,
3080383                             (unsigned int)
3080384                             allocated_text,
3080385                             process_domain_text);
3080386                 }
3080387               mb_free (allocated_data, process_domain_data);
3080388               if (DEBUG)
3080389                 {
3080390                   k_printf ("%s:%i:mb_free(%i, %zi)",
3080391                             __FILE__, __LINE__,
3080392                             (unsigned int)
3080393                             allocated_data,
3080394                             process_domain_data);
3080395                 }
3080396               inode_put (inode);
3080397               errset (EIO);
3080398               return (-1);
3080399             }
3080400           //
3080401           // Copy inside the right position to be
3080402           // executed.
3080403           //
3080404           dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080405                   memory_start, buffer,
3080406                   (size_t) size_read, NULL);
3080407         }
3080408       for (eof = 0, memory_start = allocated_data,
3080409            off_inode = header.data_offset, size_read =
3080410            0; off_inode < inode->size && !eof;
3080411            off_inode += size_read)
3080412         {
3080413           memory_start += size_read;
3080414           //
3080415           // Read a block of memory
3080416           //
3080417           size_read =
3080418             inode_file_read (inode, off_inode, buffer,
3080419                              MEM_BLOCK_SIZE, &eof);
3080420           if (size_read < 0)
3080421             {
3080422               //
3080423               // Free memory and inode.
3080424               //
3080425               mb_free (allocated_text, process_domain_text);
3080426               if (DEBUG)
3080427                 {
3080428                   k_printf ("%s:%i:mb_free(%i, %zi)",
3080429                             __FILE__, __LINE__,
3080430                             (unsigned int)
3080431                             allocated_text,
3080432                             process_domain_text);
3080433                 }
3080434               mb_free (allocated_data, process_domain_data);
3080435               if (DEBUG)
3080436                 {
3080437                   k_printf ("%s:%i:mb_free(%i, %zi)",
3080438                             __FILE__, __LINE__,
3080439                             (unsigned int)
3080440                             allocated_data,
3080441                             process_domain_data);
3080442                 }
3080443               inode_put (inode);
3080444               errset (EIO);
3080445               return (-1);
3080446             }
3080447           dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080448                   memory_start, buffer,
3080449                   (size_t) size_read, NULL);
3080450         }
3080451     }
3080452   //
3080453   // The executable file was successfully loaded in
3080454   // memory:
3080455   // release the executable file inode.
3080456   //
3080457   inode_put (inode);
3080458   //
3080459   // Update process TEXT segment inside the GDT table.
3080460   //
3080461   gdt_segment (gdt_pid_to_segment_text (pid),
3080462                (uint32_t) allocated_text,
3080463                (uint32_t) (process_domain_text / 4096),
```

```
3080464                 1, 1, 0);
3080465     //
3080466     // Update process DATA segment inside the GDT table.
3080467     //
3080468     if (process_domain_data > 0)
3080469       {
3080470         gdt_segment (gdt_pid_to_segment_data (pid),
3080471                      (uint32_t) allocated_data,
3080472                      (uint32_t) (process_domain_data /
3080473                                  4096), 1, 0, 0);
3080474       }
3080475     else
3080476       {
3080477         gdt_segment (gdt_pid_to_segment_data (pid),
3080478                      (uint32_t) allocated_text,
3080479                      (uint32_t) (process_domain_text /
3080480                                  4096), 1, 0, 0);
3080481       }
3080482     //
3080483     // Calculate segment descriptors.
3080484     //
3080485     segment_text = (gdt_pid_to_segment_text (pid) << 3) + 0;
3080486     segment_data = (gdt_pid_to_segment_data (pid) << 3) + 0;
3080487     //
3080488     // Where is the stack?
3080489     //
3080490     if (process_domain_data > 0)
3080491       {
3080492         stack_location = allocated_data;
3080493       }
3080494     else
3080495       {
3080496         stack_location = allocated_text;
3080497       }
3080498     //
3080499     // Put environment data inside the stack.
3080500     //
3080501     new_sp -= env_data_size;      // --------------
3080502     // ENVIRONMENT
3080503     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080504             (off_t) (stack_location + new_sp),
3080505             env_data, env_data_size, NULL);
3080506     //
3080507     // Put arguments data inside the stack.
3080508     //
3080509     new_sp -= arg_data_size;      // ---------------
3080510     // ARGUMENTS
3080511     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080512             (off_t) (stack_location + new_sp),
3080513             arg_data, arg_data_size, NULL);
3080514     //
3080515     // Put envp[] inside the stack, updating all the
3080516     // pointers.
3080517     //
3080518     new_sp -= 4;  // -------------------------------- NULL
3080519     stack_element = (uint32_t) NULL;
3080520     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080521             (off_t) (stack_location + new_sp),
3080522             &stack_element, (sizeof stack_element), NULL);
3080523     //
3080524     // Calculate memory pointers from original relative
3080525     // pointers, inside the environment array of
3080526     // pointers.
3080527     //
3080528     p_off = new_sp;
3080529     p_off += 4;
3080530     p_off += arg_data_size;
3080531     for (i = 0; i < envc; i++)
3080532       {
3080533         envp[i] += p_off;
3080534       }
3080535     //
3080536     new_sp -= (envc * (sizeof (char *))); // ----- *envp[]
3080537     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080538             (off_t) (stack_location + new_sp),
3080539             envp, (envc * (sizeof (char *))), NULL);
3080540     //
3080541     // Save the envp[] location, needed in the
3080542     // following.
3080543     //
3080544     envp_address = new_sp;
3080545     //
3080546     // Put argv[] inside the stack, updating all the
3080547     // pointers.
3080548     //
3080549     new_sp -= 4;  // -------------------------------- NULL
3080550     stack_element = (uint32_t) NULL;
```

```
3080551     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080552             (off_t) (stack_location + new_sp),
3080553             &stack_element, (sizeof stack_element), NULL);
3080554     //
3080555     // Calculate memory pointers from original relative
3080556     // pointers, inside the arguments array of pointers.
3080557     //
3080558     p_off = new_sp;
3080559     p_off += 4;
3080560     p_off += (envc * (sizeof (char *)));
3080561     p_off += 4;
3080562     for (i = 0; i < argc; i++)
3080563       {
3080564         argv[i] += p_off;
3080565       }
3080566     //
3080567     new_sp -= (argc * (sizeof (char *))); // ----- *argv[]
3080568     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080569             (off_t) (stack_location + new_sp),
3080570             argv, (argc * (sizeof (char *))), NULL);
3080571     //
3080572     // Save the argv[] location, needed in the
3080573     // following.
3080574     //
3080575     argv_address = new_sp;
3080576     //
3080577     // Put the pointer to the array envp[].
3080578     //
3080579     new_sp -= 4;  // ------------------------------ argc
3080580     stack_element = envp_address;
3080581     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080582             (off_t) (stack_location + new_sp),
3080583             &stack_element, (sizeof stack_element), NULL);
3080584     //
3080585     // Put the pointer to the array argv[].
3080586     //
3080587     new_sp -= 4;  // ------------------------------ argc
3080588     stack_element = argv_address;
3080589     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080590             (off_t) (stack_location + new_sp),
3080591             &stack_element, (sizeof stack_element), NULL);
3080592     //
3080593     // Put argc inside the stack.
3080594     //
3080595     new_sp -= 4;  // ------------------------------ argc
3080596     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080597             (off_t) (stack_location + new_sp),
3080598             &argc, (sizeof argc), NULL);
3080599     //
3080600     // Set the rest of the stack.
3080601     //
3080602     new_sp -= 4;  // ---------------------------- EFLAGS
3080603     stack_element = 0x0200;
3080604     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080605             (off_t) (stack_location + new_sp),
3080606             &stack_element, (sizeof stack_element), NULL);
3080607     new_sp -= 4;  // -------------------------------- CS
3080608     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080609             (off_t) (stack_location + new_sp),
3080610             &segment_text, (sizeof segment_text), NULL);
3080611     new_sp -= 4;  // ------------------------------ EIP
3080612     stack_element = 0;
3080613     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080614             (off_t) (stack_location + new_sp),
3080615             &stack_element, (sizeof stack_element), NULL);
3080616     new_sp -= 4;  // -------------------------------- GS
3080617     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080618             (off_t) (stack_location + new_sp),
3080619             &segment_data, (sizeof segment_data), NULL);
3080620     new_sp -= 4;  // -------------------------------- FS
3080621     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080622             (off_t) (stack_location + new_sp),
3080623             &segment_data, (sizeof segment_data), NULL);
3080624     new_sp -= 4;  // -------------------------------- ES
3080625     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080626             (off_t) (stack_location + new_sp),
3080627             &segment_data, (sizeof segment_data), NULL);
3080628     new_sp -= 4;  // -------------------------------- DS
3080629     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080630             (off_t) (stack_location + new_sp),
3080631             &segment_data, (sizeof segment_data), NULL);
3080632     new_sp -= 4;  // -------------------------------- EDI
3080633     stack_element = 0;
3080634     dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080635             (off_t) (stack_location + new_sp),
3080636             &stack_element, (sizeof stack_element), NULL);
3080637     new_sp -= 4;  // -------------------------------- ESI
```

```
3080638   stack_element = 0;
3080639   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080640           (off_t) (stack_location + new_sp),
3080641           &stack_element, (sizeof stack_element), NULL);
3080642   new_sp -= 4;  // ------------------------------- EBP
3080643   stack_element = 0;
3080644   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080645           (off_t) (stack_location + new_sp),
3080646           &stack_element, (sizeof stack_element), NULL);
3080647   new_sp -= 4;  // ------------------------------- EBX
3080648   stack_element = 0;
3080649   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080650           (off_t) (stack_location + new_sp),
3080651           &stack_element, (sizeof stack_element), NULL);
3080652   new_sp -= 4;  // ------------------------------- EDX
3080653   stack_element = 0;
3080654   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080655           (off_t) (stack_location + new_sp),
3080656           &stack_element, (sizeof stack_element), NULL);
3080657   new_sp -= 4;  // ------------------------------- ECX
3080658   stack_element = 0;
3080659   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080660           (off_t) (stack_location + new_sp),
3080661           &stack_element, (sizeof stack_element), NULL);
3080662   new_sp -= 4;  // ------------------------------- EAX
3080663   stack_element = 0;
3080664   dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
3080665           (off_t) (stack_location + new_sp),
3080666           &stack_element, (sizeof stack_element), NULL);
3080667   //
3080668   // Close process file descriptors, if the
3080669   // 'FD_CLOEXEC' flag
3080670   // is present.
3080671   //
3080672   for (fdn = 0; fdn < OPEN_MAX; fdn++)
3080673     {
3080674       if (proc_table[pid].fd[0].file != NULL)
3080675         {
3080676           if (proc_table[pid].fd[0].fd_flags & FD_CLOEXEC)
3080677             {
3080678               s_close (pid, fdn);
3080679             }
3080680         }
3080681     }
3080682   //
3080683   // Select device for standard I/O, if a standard I/O
3080684   // stream must be
3080685   // opened.
3080686   //
3080687   if (proc_table[pid].device_tty != 0)
3080688     {
3080689       device = proc_table[pid].device_tty;
3080690     }
3080691   else
3080692     {
3080693       device = DEV_TTY;
3080694     }
3080695   //
3080696   // Prepare missing standard file descriptors. The
3080697   // function
3080698   // 'file_stdio_dev_make()' arranges the value for
3080699   // 'errno' if
3080700   // necessary. If a standard file descriptor cannot
3080701   // be allocated,
3080702   // the program is left without it.
3080703   //
3080704   if (proc_table[pid].fd[0].file == NULL)
3080705     {
3080706       file =
3080707         file_stdio_dev_make (device, S_IFCHR, O_RDONLY);
3080708       if (file != NULL) // stdin
3080709         {
3080710           proc_table[pid].fd[0].fl_flags = O_RDONLY;
3080711           proc_table[pid].fd[0].fd_flags = 0;
3080712           proc_table[pid].fd[0].file = file;
3080713           proc_table[pid].fd[0].file->offset = 0;
3080714         }
3080715     }
3080716   if (proc_table[pid].fd[1].file == NULL)
3080717     {
3080718       file =
3080719         file_stdio_dev_make (device, S_IFCHR, O_WRONLY);
3080720       if (file != NULL) // stdout
3080721         {
3080722           proc_table[pid].fd[1].fl_flags = O_WRONLY;
3080723           proc_table[pid].fd[1].fd_flags = 0;
3080724           proc_table[pid].fd[1].file = file;
```

```
3080725           proc_table[pid].fd[1].file->offset = 0;
3080726         }
3080727     }
3080728   if (proc_table[pid].fd[2].file == NULL)
3080729     {
3080730       file =
3080731         file_stdio_dev_make (device, S_IFCHR, O_WRONLY);
3080732       if (file != NULL) // stderr
3080733         {
3080734           proc_table[pid].fd[2].fl_flags = O_WRONLY;
3080735           proc_table[pid].fd[2].fd_flags = 0;
3080736           proc_table[pid].fd[2].file = file;
3080737           proc_table[pid].fd[2].file->offset = 0;
3080738         }
3080739     }
3080740   //
3080741   // Prepare to switch
3080742   //
3080743   previous_address_text = proc_table[pid].address_text;
3080744   previous_domain_text = proc_table[pid].domain_text;
3080745   previous_address_data = proc_table[pid].address_data;
3080746   previous_domain_data = proc_table[pid].domain_data;
3080747   previous_domain_stack = proc_table[pid].domain_stack;
3080748   previous_extra_data = proc_table[pid].extra_data;
3080749   //
3080750   proc_table[pid].address_text = allocated_text;
3080751   proc_table[pid].domain_text = process_domain_text;
3080752   proc_table[pid].address_data = allocated_data;
3080753   proc_table[pid].domain_data = process_domain_data;
3080754   proc_table[pid].domain_stack = process_domain_stack;
3080755   proc_table[pid].extra_data = (size_t) 0;
3080756   proc_table[pid].sp = new_sp;
3080757   strncpy (proc_table[pid].name, path, PATH_MAX);
3080758   //
3080759   // Ensure to have a terminated string.
3080760   //
3080761   proc_table[pid].name[PATH_MAX - 1] = 0;
3080762   //
3080763   // Reset 'sig_handler[]'.
3080764   //
3080765   for (sig = 0; sig < MAX_SIGNALS; sig++)
3080766     {
3080767       proc_table[pid].sig_handler[sig] = (uintptr_t) NULL;
3080768     }
3080769   //
3080770   // Free previous data memory (included stack).
3080771   //
3080772   if (previous_domain_data > 0)
3080773     {
3080774       mb_free (previous_address_data,
3080775                previous_domain_data + previous_extra_data);
3080776       if (DEBUG)
3080777         {
3080778           k_printf ("%s:%i:mb_free(%i, %zi)",
3080779                     __FILE__, __LINE__,
3080780                     (unsigned int)
3080781                     previous_address_data,
3080782                     previous_domain_data +
3080783                     previous_extra_data);
3080784         }
3080785     }
3080786   //
3080787   // Free code memory if not shared.
3080788   //
3080789   for (proc_count = 0, extra = 0; extra < PROCESS_MAX;
3080790        extra++)
3080791     {
3080792       if (proc_table[extra].status == PROC_EMPTY ||
3080793           proc_table[extra].status == PROC_ZOMBIE)
3080794         {
3080795           continue;
3080796         }
3080797       if (previous_address_text ==
3080798           proc_table[extra].address_text)
3080799         {
3080800           proc_count++;
3080801         }
3080802     }
3080803   if (proc_count == 0)
3080804     {
3080805       //
3080806       // The code segment can be released, because no
3080807       // other
3080808       // process is using it.
3080809       //
3080810       if (previous_domain_data > 0)
3080811         {
```

```
3080812           mb_free (previous_address_text,
3080813                    previous_domain_text +
3080814                    previous_extra_data);
3080815           if (DEBUG)
3080816             {
3080817               k_printf ("%s:%i:mb_free(%i, %zi)",
3080818                         __FILE__, __LINE__,
3080819                         (unsigned int)
3080820                         previous_address_text,
3080821                         previous_domain_text +
3080822                         previous_extra_data);
3080823             }
3080824         }
3080825       else
3080826         {
3080827           mb_free (previous_address_text,
3080828                    previous_domain_text);
3080829           if (DEBUG)
3080830             {
3080831               k_printf ("%s:%i:mb_free(%i, %zi)",
3080832                         __FILE__, __LINE__,
3080833                         (unsigned int)
3080834                         previous_address_text,
3080835                         previous_domain_text);
3080836             }
3080837         }
3080838     }
3080839   //
3080840   // Change the segment and the stack pointer, from
3080841   // the interrupt.
3080842   // [1] Anyway,the stack segment selector does not
3080843   // change.
3080844   //
3080845   proc_stack_segment_selector = segment_data;   // [1]
3080846   proc_stack_pointer = proc_table[pid].sp;
3080847   //
3080848   //
3080849   //
3080850   return (0);
3080851 }
```

94.14.23 kernel/proc/proc_timer_init.c

Si veda la sezione 93.20.22.

```
3090001 #include <kernel/proc.h>
3090002 #include <stdint.h>
3090003 #include <kernel/lib_k.h>
3090004 #include <kernel/ibm_i386.h>
3090005 #include <stdint.h>
3090006 //-----------------------------------------------------------
3090007 void
3090008 proc_timer_init (clock_t freq)
3090009 {
3090010   int input_freq = 1193180;
3090011   //
3090012   // La frequenza di riferimento è 1,19318 MHz, la
3090013   // quale va
3090014   // divisa per la frequenza che si intende avere
3090015   // effettivamente.
3090016   //
3090017   int divisor = input_freq / freq;
3090018   //
3090019   // Il risultato deve essere un valore intero
3090020   // maggiore di zero
3090021   // e inferiore di UINT16_MAX, altrimenti è stata
3090022   // chiesta una
3090023   // frequenza troppo elevata o troppo bassa.
3090024   //
3090025   if (divisor == 0 || divisor > UINT16_MAX)
3090026     {
3090027       k_printf
3090028         ("[%s] ERROR: IRQ 0 frequency wrong: %i Hz!\n"
3090029          "[%s]        The min allowed frequency \n"
3090030          "[%s]        is 18.22 Hz.\n",
3090031          "[%s]        The max allowed frequency \n"
3090032          "[%s]        is 1.19 MHz.\n",
3090033          __func__, freq, __func__, __func__, __func__,
3090034          __func__);
3090035       return;
3090036     }
3090037   //
3090038   // Il valore che si ottiene, ovvero il «divisore»,
3090039   // va
3090040   // comunicato al PIT (programmable interval timer),
3090041   // spezzandolo in due parti.
3090042   //
```



```
3090043   out_8 ((uint32_t) 0x43, (uint32_t) 0x36);
3090044   //
3090045   // Lower byte.
3090046   //
3090047   out_8 ((uint32_t) 0x40, (uint32_t) (divisor & 0x0F));
3090048   //
3090049   // Higher byte.
3090050   //
3090051   out_8 ((uint32_t) 0x40, (uint32_t) (divisor / 0x10));
3090052 }
```

94.14.24 kernel/proc/proc_wakeup_pipe_read.c

Si veda la sezione 93.20.23.

```
3100001 #include <kernel/proc.h>
3100002 //-----------------------------------------------------------
3100003 void
3100004 proc_wakeup_pipe_read (inode_t * inode)
3100005 {
3100006   pid_t pid;
3100007
3100008   for (pid = 1; pid < PROCESS_MAX; pid++)
3100009     {
3100010       if ((proc_table[pid].status == PROC_SLEEPING)
3100011           && (proc_table[pid].wakeup_events
3100012               & WAKEUP_EVENT_PIPE_READ)
3100013           && (proc_table[pid].wakeup_inode == inode))
3100014         {
3100015           proc_table[pid].wakeup_events = 0;
3100016           proc_table[pid].wakeup_inode = NULL;
3100017           proc_table[pid].status = PROC_READY;
3100018         }
3100019     }
3100020 }
```

94.14.25 kernel/proc/proc_wakeup_pipe_write.c

Si veda la sezione 93.20.23.

```
3110001 #include <kernel/proc.h>
3110002 //-----------------------------------------------------------
3110003 void
3110004 proc_wakeup_pipe_write (inode_t * inode)
3110005 {
3110006   pid_t pid;
3110007
3110008   for (pid = 1; pid < PROCESS_MAX; pid++)
3110009     {
3110010       if ((proc_table[pid].status == PROC_SLEEPING)
3110011           && (proc_table[pid].wakeup_events
3110012               & WAKEUP_EVENT_PIPE_WRITE)
3110013           && (proc_table[pid].wakeup_inode == inode))
3110014         {
3110015           proc_table[pid].wakeup_events = 0;
3110016           proc_table[pid].wakeup_inode = NULL;
3110017           proc_table[pid].status = PROC_READY;
3110018         }
3110019     }
3110020 }
```

94.14.26 kernel/proc/proc_wakeup_terminal.c

Si veda la sezione 93.20.23.

```
3120001 #include <kernel/proc.h>
3120002 #include <kernel/lib_k.h>
3120003 #include <sys/types.h>
3120004 //-----------------------------------------------------------
3120005 void
3120006 proc_wakeup_terminal (void)
3120007 {
3120008   pid_t pid;
3120009   int maj;
3120010   //
3120011   // At the moment, all processes waiting for reading
3120012   // a terminal
3120013   // or the console are reactivated.
3120014   //
3120015   for (pid = 0; pid < PROCESS_MAX; pid++)
3120016     {
3120017       if ((proc_table[pid].status == PROC_SLEEPING)
3120018           && (proc_table[pid].wakeup_events &
3120019               WAKEUP_EVENT_DEV_READ))
3120020         {
3120021           maj = major (proc_table[pid].wakeup_dev);
3120022           if (maj == DEV_TTY_MAJOR
```

```
3120023                 || maj == DEV_CONSOLE_MAJOR)
3120024               {
3120025                 //
3120026                 // A process waiting for that terminal
3120027                 // was found:
3120028                 // remove the waiting event and set it
3120029                 // ready.
3120030                 //
3120031                 proc_table[pid].wakeup_events &=
3120032                   ~WAKEUP_EVENT_DEV_READ;
3120033                 proc_table[pid].wakeup_dev = 0;
3120034                 proc_table[pid].status = PROC_READY;
3120035               }
3120036           }
3120037       }
3120038   }
```

94.14.27 kernel/proc/ptr.c

«

Si veda la sezione 93.20.27.

```
3130001 #include <kernel/proc.h>
3130002 #include <kernel/lib_s.h>
3130003 #include <kernel/lib_k.h>
3130004 #include <stdint.h>
3130005 //-----------------------------------------------------------
3130006 #define DEBUG 0
3130007 //-----------------------------------------------------------
3130008 void *
3130009 ptr (pid_t pid, void *p)
3130010 {
3130011   uintptr_t start;
3130012   //
3130013   if (p == NULL)
3130014     {
3130015       return (NULL);
3130016     }
3130017   else if (proc_table[pid].domain_data == 0)
3130018     {
3130019       start = proc_table[pid].address_text;
3130020     }
3130021   else
3130022     {
3130023       start = proc_table[pid].address_data;
3130024     }
3130025   //
3130026   return ((void *) (start + (uintptr_t) p));
3130027 }
```

94.14.28 kernel/proc/sysroutine.c

«

Si veda la sezione 93.20.28.

```
3140001 #include <kernel/proc.h>
3140002 #include <errno.h>
3140003 #include <kernel/lib_k.h>
3140004 #include <kernel/lib_s.h>
3140005 #include <stdint.h>
3140006 //-----------------------------------------------------------
3140007 static void sysroutine_error_back (int *number,
3140008                                    int *line,
3140009                                    char *file_name);
3140010 //-----------------------------------------------------------
3140011 void
3140012 sysroutine (uint32_t syscallnr, uint32_t msg_off,
3140013             uint32_t msg_size)
3140014 {
3140015   pid_t pid = proc_current;
3140016   //
3140017   // Inbox.
3140018   //
3140019   union
3140020   {
3140021     sysmsg_accept_t accept;
3140022     sysmsg_bind_t bind;
3140023     sysmsg_brk_t brk;
3140024     sysmsg_chdir_t chdir;
3140025     sysmsg_chmod_t chmod;
3140026     sysmsg_chown_t chown;
3140027     sysmsg_clock_t clock;
3140028     sysmsg_close_t close;
3140029     sysmsg_connect_t connect;
3140030     sysmsg_dup_t dup;
3140031     sysmsg_dup2_t dup2;
3140032     sysmsg_exec_t exec;
3140033     sysmsg_exit_t exit;
3140034     sysmsg_fchmod_t fchmod;
```



```
3140035     sysmsg_fchown_t fchown;
3140036     sysmsg_fcntl_t fcntl;
3140037     sysmsg_fork_t fork;
3140038     sysmsg_fstat_t fstat;
3140039     sysmsg_ipconfig_t ipconfig;
3140040     sysmsg_jmp_t jmp;
3140041     sysmsg_kill_t kill;
3140042     sysmsg_link_t link;
3140043     sysmsg_listen_t listen;
3140044     sysmsg_lseek_t lseek;
3140045     sysmsg_mkdir_t mkdir;
3140046     sysmsg_mknod_t mknod;
3140047     sysmsg_mount_t mount;
3140048     sysmsg_open_t open;
3140049     sysmsg_pipe_t pipe;
3140050     sysmsg_read_t read;
3140051     sysmsg_recvfrom_t recvfrom;
3140052     sysmsg_route_t route;
3140053     sysmsg_send_t send;
3140054     sysmsg_sbrk_t sbrk;
3140055     sysmsg_seteuid_t seteuid;
3140056     sysmsg_setuid_t setuid;
3140057     sysmsg_setegid_t setegid;
3140058     sysmsg_setgid_t setgid;
3140059     sysmsg_signal_t signal;
3140060     sysmsg_sleep_t sleep;
3140061     sysmsg_socket_t socket;
3140062     sysmsg_stat_t stat;
3140063     sysmsg_stime_t stime;
3140064     sysmsg_tcattr_t tcattr;
3140065     sysmsg_time_t time;
3140066     sysmsg_uarea_t uarea;
3140067     sysmsg_umask_t umask;
3140068     sysmsg_umount_t umount;
3140069     sysmsg_unlink_t unlink;
3140070     sysmsg_wait_t wait;
3140071     sysmsg_write_t write;
3140072     sysmsg_zpchar_t zpchar;
3140073     sysmsg_zpstring_t zpstring;
3140074   } *msg;
3140075   //
3140076   // Align the message address pointer to the source
3140077   // message.
3140078   //
3140079   msg = ptr (pid, (void *) msg_off);
3140080   //
3140081   // Verify if the system call was emitted from kernel
3140082   // code.
3140083   // The kernel can emit only some particular system
3140084   // call:
3140085   // SYS_NULL, to let other processes run;
3140086   // SYS_FORK, to let fork itself;
3140087   // SYS_EXEC, to replace a forked copy of itself.
3140088   //
3140089   if (pid == 0)
3140090     {
3140091       //
3140092       // This is the kernel code!
3140093       //
3140094       if (syscallnr != SYS_0
3140095           && syscallnr != SYS_FORK
3140096           && syscallnr != SYS_EXEC
3140097           && syscallnr != SYS_ZPSTRING)
3140098         {
3140099           k_printf
3140100             ("kernel panic: the system call %i ",
3140101              syscallnr);
3140102           k_printf
3140103             ("was received while running "
3140104              "in kernel space!\n");
3140105         }
3140106     }
3140107   //
3140108   // Entering a system call, the kernel variable
3140109   // 'errno' must be
3140110   // reset, otherwise, a previous kernel code error
3140111   // might be returned
3140112   // to the applications.
3140113   //
3140114   errno = 0;
3140115   errln = 0;
3140116   errfn[0] = 0;
3140117   //
3140118   // Do the request from the received system call.
3140119   //
3140120   switch (syscallnr)
3140121     {
```

```
3140122     case SYS_0:
3140123       break;
3140124     case SYS_ACCEPT:
3140125       msg->accept.ret =
3140126         s_accept (pid, msg->accept.sfdn,
3140127                   &msg->accept.addr, &msg->accept.addrlen);
3140128       msg->accept.fl_flags =
3140129         proc_table[pid].fd[msg->accept.sfdn].fl_flags;
3140130       sysroutine_error_back (&msg->accept.errno,
3140131                              &msg->accept.errln,
3140132                              msg->accept.errfn);
3140133       break;
3140134     case SYS_BIND:
3140135       msg->bind.ret = s_bind (pid, msg->bind.sfdn,
3140136                               &msg->bind.addr,
3140137                               msg->bind.addrlen);
3140138       sysroutine_error_back (&msg->bind.errno,
3140139                              &msg->bind.errln,
3140140                              msg->bind.errfn);
3140141       break;
3140142     case SYS_BRK:
3140143       msg->brk.ret = s_brk (pid, msg->brk.address);
3140144       sysroutine_error_back (&msg->brk.errno,
3140145                              &msg->brk.errln,
3140146                              msg->brk.errfn);
3140147       break;
3140148     case SYS_CHDIR:
3140149       msg->chdir.ret =
3140150         s_chdir (pid, ptr (pid, (void *) msg->chdir.path));
3140151       sysroutine_error_back (&msg->chdir.errno,
3140152                              &msg->chdir.errln,
3140153                              msg->chdir.errfn);
3140154       break;
3140155     case SYS_CHMOD:
3140156       msg->chmod.ret = s_chmod (pid,
3140157                                 ptr (pid,
3140158                                      (void *) msg->
3140159                                      chmod.path),
3140160                                 msg->chmod.mode);
3140161       sysroutine_error_back (&msg->chmod.errno,
3140162                              &msg->chmod.errln,
3140163                              msg->chmod.errfn);
3140164       break;
3140165     case SYS_CHOWN:
3140166       msg->chown.ret = s_chown (pid,
3140167                                 ptr (pid,
3140168                                      (void *) msg->
3140169                                      chown.path),
3140170                                 msg->chown.uid,
3140171                                 msg->chown.gid);
3140172       sysroutine_error_back (&msg->chown.errno,
3140173                              &msg->chown.errln,
3140174                              msg->chown.errfn);
3140175       break;
3140176     case SYS_CLOCK:
3140177       msg->clock.ret = s_clock (pid);
3140178       break;
3140179     case SYS_CLOSE:
3140180       msg->close.ret = s_close (pid, msg->close.fdn);
3140181       sysroutine_error_back (&msg->close.errno,
3140182                              &msg->close.errln,
3140183                              msg->close.errfn);
3140184       break;
3140185     case SYS_CONNECT:
3140186       msg->connect.ret =
3140187         s_connect (pid, msg->connect.sfdn,
3140188                    &msg->connect.addr,
3140189                    msg->connect.addrlen);
3140190       sysroutine_error_back (&msg->connect.errno,
3140191                              &msg->connect.errln,
3140192                              msg->connect.errfn);
3140193       break;
3140194     case SYS_DUP:
3140195       msg->dup.ret = s_dup (pid, msg->dup.fdn_old);
3140196       sysroutine_error_back (&msg->dup.errno,
3140197                              &msg->dup.errln,
3140198                              msg->dup.errfn);
3140199       break;
3140200     case SYS_DUP2:
3140201       msg->dup2.ret = s_dup2 (pid, msg->dup2.fdn_old,
3140202                               msg->dup2.fdn_new);
3140203       sysroutine_error_back (&msg->dup2.errno,
3140204                              &msg->dup2.errln,
3140205                              msg->dup2.errfn);
3140206       break;
3140207     case SYS_EXEC:
3140208       msg->exec.ret = proc_sys_exec (pid,
```

```
3140209                                      ptr (pid,
3140210                                           (void *)
3140211                                           msg->exec.path),
3140212                                      msg->exec.argc,
3140213                                      msg->exec.arg_data,
3140214                                      msg->exec.envc,
3140215                                      msg->exec.env_data);
3140216       msg->exec.uid = proc_table[pid].uid;
3140217       msg->exec.euid = proc_table[pid].euid;
3140218       sysroutine_error_back (&msg->exec.errno,
3140219                              &msg->exec.errln,
3140220                              msg->exec.errfn);
3140221       break;
3140222     case SYS_EXIT:
3140223       if (pid == 0)
3140224         {
3140225           k_printf ("kernel alert: "
3140226                     "the kernel cannot exit!\n");
3140227         }
3140228       else
3140229         {
3140230           s__exit (pid, msg->exit.status);
3140231         }
3140232       break;
3140233     case SYS_FCHMOD:
3140234       msg->fchmod.ret = s_fchmod (pid, msg->fchmod.fdn,
3140235                                   msg->fchmod.mode);
3140236       sysroutine_error_back (&msg->fchmod.errno,
3140237                              &msg->fchmod.errln,
3140238                              msg->fchmod.errfn);
3140239       break;
3140240     case SYS_FCHOWN:
3140241       msg->fchown.ret = s_fchown (pid, msg->fchown.fdn,
3140242                                   msg->fchown.uid,
3140243                                   msg->fchown.gid);
3140244       sysroutine_error_back (&msg->fchown.errno,
3140245                              &msg->fchown.errln,
3140246                              msg->fchown.errfn);
3140247       break;
3140248     case SYS_FCNTL:
3140249       msg->fcntl.ret = s_fcntl (pid, msg->fcntl.fdn,
3140250                                 msg->fcntl.cmd,
3140251                                 msg->fcntl.arg);
3140252       sysroutine_error_back (&msg->fcntl.errno,
3140253                              &msg->fcntl.errln,
3140254                              msg->fcntl.errfn);
3140255       break;
3140256     case SYS_FORK:
3140257       msg->fork.ret = s_fork (pid);
3140258       sysroutine_error_back (&msg->fork.errno,
3140259                              &msg->fork.errln,
3140260                              msg->fork.errfn);
3140261       break;
3140262     case SYS_FSTAT:
3140263       msg->fstat.ret =
3140264         s_fstat (pid, msg->fstat.fdn, &msg->fstat.stat);
3140265       sysroutine_error_back (&msg->fstat.errno,
3140266                              &msg->fstat.errln,
3140267                              msg->fstat.errfn);
3140268       break;
3140269     case SYS_IPCONFIG:
3140270       msg->ipconfig.ret =
3140271         s_ipconfig (pid, msg->ipconfig.n,
3140272                     msg->ipconfig.address, msg->ipconfig.m);
3140273       sysroutine_error_back (&msg->ipconfig.errno,
3140274                              &msg->ipconfig.errln,
3140275                              msg->ipconfig.errfn);
3140276       break;
3140277     case SYS_KILL:
3140278       msg->kill.ret =
3140279         s_kill (pid, msg->kill.pid, msg->kill.signal);
3140280       sysroutine_error_back (&msg->kill.errno,
3140281                              &msg->kill.errln,
3140282                              msg->kill.errfn);
3140283       break;
3140284     case SYS_LINK:
3140285       msg->link.ret
3140286         = s_link (pid,
3140287                   ptr (pid,
3140288                        (void *) msg->link.path_old),
3140289                   ptr (pid, (void *) msg->link.path_new));
3140290       sysroutine_error_back (&msg->link.errno,
3140291                              &msg->link.errln,
3140292                              msg->link.errfn);
3140293       break;
3140294     case SYS_LISTEN:
3140295       msg->listen.ret =
```

```
3140296         s_listen (pid, msg->listen.sfdn,
3140297                   msg->listen.backlog);
3140298       sysroutine_error_back (&msg->listen.errno,
3140299                              &msg->listen.errln,
3140300                              msg->listen.errfn);
3140301       break;
3140302     case SYS_LONGJMP:
3140303       s_longjmp (pid, msg->jmp.env, msg->jmp.ret);
3140304       break;
3140305     case SYS_LSEEK:
3140306       msg->lseek.ret = s_lseek (pid, msg->lseek.fdn,
3140307                                 msg->lseek.offset,
3140308                                 msg->lseek.whence);
3140309       sysroutine_error_back (&msg->lseek.errno,
3140310                              &msg->lseek.errln,
3140311                              msg->lseek.errfn);
3140312       break;
3140313     case SYS_MKDIR:
3140314       msg->mkdir.ret = s_mkdir (pid,
3140315                                 ptr (pid,
3140316                                      (void *) msg->
3140317                                      mkdir.path),
3140318                                 msg->mkdir.mode);
3140319       sysroutine_error_back (&msg->mkdir.errno,
3140320                              &msg->mkdir.errln,
3140321                              msg->mkdir.errfn);
3140322       break;
3140323     case SYS_MKNOD:
3140324       msg->mknod.ret = s_mknod (pid,
3140325                                 ptr (pid,
3140326                                      (void *) msg->
3140327                                      mknod.path),
3140328                                 msg->mknod.mode,
3140329                                 msg->mknod.device);
3140330       sysroutine_error_back (&msg->mknod.errno,
3140331                              &msg->mknod.errln,
3140332                              msg->mknod.errfn);
3140333       break;
3140334     case SYS_MOUNT:
3140335       msg->mount.ret = s_mount (pid,
3140336                                 ptr (pid,
3140337                                      (void *) msg->
3140338                                      mount.path_dev),
3140339                                 ptr (pid,
3140340                                      (void *) msg->
3140341                                      mount.path_mnt),
3140342                                 msg->mount.options);
3140343       sysroutine_error_back (&msg->mount.errno,
3140344                              &msg->mount.errln,
3140345                              msg->mount.errfn);
3140346       break;
3140347     case SYS_OPEN:
3140348       msg->open.ret = s_open (pid,
3140349                               ptr (pid,
3140350                                    (void *) msg->open.path),
3140351                               msg->open.flags,
3140352                               msg->open.mode);
3140353       sysroutine_error_back (&msg->open.errno,
3140354                              &msg->open.errln,
3140355                              msg->open.errfn);
3140356       break;
3140357     case SYS_PIPE:
3140358       msg->pipe.ret = s_pipe (pid, msg->pipe.pipefd);
3140359       sysroutine_error_back (&msg->pipe.errno,
3140360                              &msg->pipe.errln,
3140361                              msg->pipe.errfn);
3140362       break;
3140363     case SYS_PGRP:
3140364       proc_table[pid].pgrp = pid;
3140365       break;
3140366     case SYS_READ:
3140367       msg->read.ret = s_read (pid, msg->read.fdn,
3140368                               ptr (pid,
3140369                                    msg->read.buffer),
3140370                               msg->read.count);
3140371       msg->read.fl_flags =
3140372         proc_table[pid].fd[msg->read.fdn].fl_flags;
3140373       sysroutine_error_back (&msg->read.errno,
3140374                              &msg->read.errln,
3140375                              msg->read.errfn);
3140376       break;
3140377     case SYS_RECVFROM:
3140378       msg->recvfrom.ret =
3140379         s_recvfrom
3140380         (pid, msg->recvfrom.sfdn,
3140381          ptr (pid, msg->recvfrom.buffer),
3140382          msg->recvfrom.count,
```



```
3140383          msg->recvfrom.flags, ptr (pid,
3140384                                    msg->recvfrom.addrfrom),
3140385          ptr (pid, msg->recvfrom.addrsize));
3140386       msg->recvfrom.fl_flags =
3140387         proc_table[pid].fd[msg->recvfrom.sfdn].fl_flags;
3140388       sysroutine_error_back (&msg->recvfrom.errno,
3140389                              &msg->recvfrom.errln,
3140390                              msg->recvfrom.errfn);
3140391       break;
3140392     case SYS_ROUTEADD:
3140393       msg->route.ret =
3140394         s_routeadd (pid, msg->route.destination,
3140395                     msg->route.m, msg->route.router,
3140396                     msg->route.device);
3140397       sysroutine_error_back (&msg->route.errno,
3140398                              &msg->route.errln,
3140399                              msg->route.errfn);
3140400       break;
3140401     case SYS_ROUTEDEL:
3140402       msg->route.ret =
3140403         s_routedel (pid, msg->route.destination,
3140404                     msg->route.m);
3140405       sysroutine_error_back (&msg->route.errno,
3140406                              &msg->route.errln,
3140407                              msg->route.errfn);
3140408       break;
3140409     case SYS_SBRK:
3140410       msg->sbrk.ret = s_sbrk (pid, msg->sbrk.increment);
3140411       sysroutine_error_back (&msg->sbrk.errno,
3140412                              &msg->sbrk.errln,
3140413                              msg->sbrk.errfn);
3140414       break;
3140415     case SYS_SEND:
3140416       msg->send.ret = s_send (pid, msg->send.sfdn,
3140417                               ptr (pid,
3140418                                    (void *) msg->send.
3140419                                    buffer), msg->send.count,
3140420                               msg->send.flags);
3140421       sysroutine_error_back (&msg->send.errno,
3140422                              &msg->send.errln,
3140423                              msg->send.errfn);
3140424       break;
3140425     case SYS_SETEUID:
3140426       msg->seteuid.ret = s_seteuid (pid, msg->seteuid.euid);
3140427       msg->seteuid.euid = proc_table[pid].euid;
3140428       sysroutine_error_back (&msg->seteuid.errno,
3140429                              &msg->seteuid.errln,
3140430                              msg->seteuid.errfn);
3140431       break;
3140432     case SYS_SETUID:
3140433       msg->setuid.ret = s_setuid (pid, msg->setuid.euid);
3140434       msg->setuid.uid = proc_table[pid].uid;
3140435       msg->setuid.euid = proc_table[pid].euid;
3140436       msg->setuid.suid = proc_table[pid].suid;
3140437       sysroutine_error_back (&msg->setuid.errno,
3140438                              &msg->setuid.errln,
3140439                              msg->setuid.errfn);
3140440       break;
3140441     case SYS_SETEGID:
3140442       msg->setegid.ret = s_setegid (pid, msg->setegid.egid);
3140443       msg->setegid.egid = proc_table[pid].egid;
3140444       sysroutine_error_back (&msg->setegid.errno,
3140445                              &msg->setegid.errln,
3140446                              msg->setegid.errfn);
3140447       break;
3140448     case SYS_SETGID:
3140449       msg->setgid.ret = s_setgid (pid, msg->setgid.egid);
3140450       msg->setgid.gid = proc_table[pid].gid;
3140451       msg->setgid.egid = proc_table[pid].egid;
3140452       msg->setgid.sgid = proc_table[pid].sgid;
3140453       sysroutine_error_back (&msg->setgid.errno,
3140454                              &msg->setgid.errln,
3140455                              msg->setgid.errfn);
3140456       break;
3140457     case SYS_SETJMP:
3140458       msg->jmp.ret = s_setjmp (pid, msg->jmp.env);
3140459       break;
3140460     case SYS_SIGNAL:
3140461       msg->signal.ret =
3140462         s_signal (pid, msg->signal.signal,
3140463                   msg->signal.handler, msg->signal.wrapper);
3140464       sysroutine_error_back (&msg->signal.errno,
3140465                              &msg->signal.errln,
3140466                              msg->signal.errfn);
3140467       break;
3140468     case SYS_SLEEP:
3140469       proc_table[pid].status = PROC_SLEEPING;
```

```
3140470       proc_table[pid].ret = 0;
3140471       proc_table[pid].wakeup_events = msg->sleep.events;
3140472       proc_table[pid].wakeup_timer = msg->sleep.seconds;
3140473       break;
3140474     case SYS_STAT:
3140475       msg->stat.ret = s_stat (pid,
3140476                               ptr (pid,
3140477                                    (void *) msg->stat.path),
3140478                               &msg->stat.stat);
3140479       sysroutine_error_back (&msg->stat.errno,
3140480                              &msg->stat.errln,
3140481                              msg->stat.errfn);
3140482       break;
3140483     case SYS_STIME:
3140484       msg->stime.ret = s_stime (pid, &msg->stime.timer);
3140485       break;
3140486     case SYS_TCGETATTR:
3140487       msg->tcattr.ret =
3140488         s_tcgetattr (pid, msg->tcattr.fdn,
3140489                      ptr (pid, msg->tcattr.attr));
3140490       sysroutine_error_back (&msg->tcattr.errno,
3140491                              &msg->tcattr.errln,
3140492                              msg->tcattr.errfn);
3140493       break;
3140494     case SYS_TCSETATTR:
3140495       msg->tcattr.ret =
3140496         s_tcsetattr (pid, msg->tcattr.fdn,
3140497                      msg->tcattr.action, ptr (pid,
3140498                                               msg->tcattr.
3140499                                               attr));
3140500       sysroutine_error_back (&msg->tcattr.errno,
3140501                              &msg->tcattr.errln,
3140502                              msg->tcattr.errfn);
3140503       break;
3140504     case SYS_TIME:
3140505       msg->time.ret = s_time (pid, NULL);
3140506       break;
3140507     case SYS_UAREA:
3140508       msg->uarea.uid = proc_table[pid].uid;
3140509       msg->uarea.suid = proc_table[pid].suid;
3140510       msg->uarea.euid = proc_table[pid].euid;
3140511       msg->uarea.gid = proc_table[pid].gid;
3140512       msg->uarea.sgid = proc_table[pid].sgid;
3140513       msg->uarea.egid = proc_table[pid].egid;
3140514       msg->uarea.pid = pid;
3140515       msg->uarea.ppid = proc_table[pid].ppid;
3140516       msg->uarea.pgrp = proc_table[pid].pgrp;
3140517       msg->uarea.umask = proc_table[pid].umask;
3140518       strncpy (ptr (pid, msg->uarea.path_cwd),
3140519                proc_table[pid].path_cwd,
3140520                msg->uarea.path_cwd_size);
3140521       break;
3140522     case SYS_UMASK:
3140523       msg->umask.ret = proc_table[pid].umask;
3140524       proc_table[pid].umask = (msg->umask.umask & 00777);
3140525       break;
3140526     case SYS_UMOUNT:
3140527       msg->umount.ret = s_umount (pid,
3140528                                   ptr (pid,
3140529                                        (void *)
3140530                                        msg->umount.
3140531                                        path_mnt));
3140532       sysroutine_error_back (&msg->umount.errno,
3140533                              &msg->umount.errln,
3140534                              msg->umount.errfn);
3140535       break;
3140536     case SYS_UNLINK:
3140537       msg->unlink.ret = s_unlink (pid,
3140538                                   ptr (pid,
3140539                                        (void *) msg->
3140540                                        unlink.path));
3140541       sysroutine_error_back (&msg->unlink.errno,
3140542                              &msg->unlink.errln,
3140543                              msg->unlink.errfn);
3140544       break;
3140545     case SYS_WAIT:
3140546       msg->wait.ret = s_wait (pid, &msg->wait.status);
3140547       sysroutine_error_back (&msg->wait.errno,
3140548                              &msg->wait.errln,
3140549                              msg->wait.errfn);
3140550       break;
3140551     case SYS_WRITE:
3140552       msg->write.ret = s_write (pid, msg->write.fdn,
3140553                                 ptr (pid,
3140554                                      (void *) msg->
3140555                                      write.buffer),
3140556                                 msg->write.count);
3140557       sysroutine_error_back (&msg->write.errno,
3140558                              &msg->write.errln,
3140559                              msg->write.errfn);
3140560       break;
3140561     case SYS_SOCKET:
3140562       msg->socket.ret =
3140563         s_socket (pid, msg->socket.family,
3140564                   msg->socket.type, msg->socket.protocol);
3140565       sysroutine_error_back (&msg->socket.errno,
3140566                              &msg->socket.errln,
3140567                              msg->socket.errfn);
3140568       break;
3140569     case SYS_ZPCHAR:
3140570       dev_io (pid, DEV_TTY, DEV_WRITE, 0L,
3140571               &msg->zpchar.c, 1, NULL);
3140572       break;
3140573     case SYS_ZPSTRING:
3140574       dev_io (pid, DEV_TTY, DEV_WRITE, (off_t) 0,
3140575               msg->zpstring.string,
3140576               strlen (msg->zpstring.string), NULL);
3140577       break;
3140578     default:
3140579       k_printf
3140580         ("kernel alert: unknown system call %i!\n",
3140581          syscallnr);
3140582       break;
3140583     }
3140584   //
3140585   // Continue with the scheduler.
3140586   //
3140587   proc_scheduler ();
3140588 }
3140589
3140590 //-------------------------------------------------------
3140591 static void
3140592 sysroutine_error_back (int *number, int *line,
3140593                        char *file_name)
3140594 {
3140595   *number = errno;
3140596   *line = errln;
3140597   strncpy (file_name, errfn, PATH_MAX);
3140598   file_name[PATH_MAX - 1] = 0;
3140599 }
```