

94	Script e sorgenti del kernel	399
94.1	os32: directory principale	404
94.2	os32: «kernel/blk.h»	416
94.3	os32: «kernel/dev.h»	420
94.4	os32: «kernel/dm.h»	429
94.5	os32: «kernel/fs.h»	475
94.6	os32: «kernel/ibm_i386.h»	530
94.7	os32: «kernel/lib_k.h»	554
94.8	os32: «kernel/lib_s.h»	558
94.9	os32: «kernel/main.h»	627
94.10	os32: «kernel/memory.h»	635
94.11	os32: «kernel/multiboot.h»	641
94.12	os32: «kernel/net.h»	644
94.13	os32: «kernel/part.h»	703
94.14	os32: «kernel/proc.h»	704
95	Sorgenti della libreria generale	749
95.1	os32: file isolati della directory «lib/»	753
95.2	os32: «lib/_gcc.h»	758
95.3	os32: «lib/arpa/inet.h»	761
95.4	os32: «lib/dirent.h»	764
95.5	os32: «lib/errno.h»	768
95.6	os32: «lib/fcntl.h»	773
95.7	os32: «lib/grp.h»	776
95.8	os32: «lib/inttypes.h»	777
95.9	os32: «lib/libgen.h»	781
95.10	os32: «lib/netinet/icmp.h»	783
95.11	os32: «lib/netinet/in.h»	785
95.12	os32: «lib/netinet/ip.h»	786
95.13	os32: «lib/netinet/tcp.h»	787
95.14	os32: «lib/netinet/udp.h»	788
95.15	os32: «lib/pwd.h»	788
95.16	os32: «lib/setjmp.h»	790
95.17	os32: «lib/signal.h»	792
95.18	os32: «lib/stdio.h»	794
95.19	os32: «lib/stdlib.h»	844
95.20	os32: «lib/string.h»	866
95.21	os32: «lib/sys/os32.h»	877
95.22	os32: «lib/sys/sa_family_t.h»	893
95.23	os32: «lib/sys/socket.h»	893
95.24	os32: «lib/sys/socklen_t.h»	899
95.25	os32: «lib/sys/stat.h»	899
95.26	os32: «lib/sys/types.h»	904
95.27	os32: «lib/sys/wait.h»	905
95.28	os32: «lib/termios.h»	905
95.29	os32: «lib/time.h»	907
95.30	os32: «lib/unistd.h»	913
95.31	os32: «lib/utime.h»	935
96	Sorgenti delle applicazioni	937
96.1	os32: directory «applic/»	938
	Indice analitico del volume	1043

## Script e sorgenti del kernel

94.1	os32: directory principale .....	404
94.1.1	applic.sep.ld .....	404
94.1.2	bochs .....	405
94.1.3	elf-to-os32 .....	405
94.1.4	fdisk .....	406
94.1.5	file_image_functions .....	407
94.1.6	format .....	410
94.1.7	kernel.ld .....	410
94.1.8	makeit.sep .....	411
94.1.9	qemu .....	415
94.1.10	syslinux .....	416
94.1.11	tap0 .....	416
94.2	os32: «kernel/blk.h» .....	416
94.2.1	kernel/blk/blk_ata.c .....	417
94.2.2	kernel/blk/blk_cache_check.c .....	418
94.2.3	kernel/blk/blk_cache_init.c .....	419
94.2.4	kernel/blk/blk_cache_read.c .....	419
94.2.5	kernel/blk/blk_cache_save.c .....	419
94.2.6	kernel/blk/blk_public.c .....	420
94.3	os32: «kernel/dev.h» .....	420
94.3.1	kernel/dev/dev_ata.c .....	421
94.3.2	kernel/dev/dev_dm.c .....	422
94.3.3	kernel/dev/dev_io.c .....	423
94.3.4	kernel/dev/dev_kmem.c .....	423
94.3.5	kernel/dev/dev_mem.c .....	426
94.3.6	kernel/dev/dev_tty.c .....	427
94.4	os32: «kernel/dm.h» .....	429
94.4.1	kernel/dm/dm_init.c .....	430
94.4.2	kernel/dm/dm_public.c .....	431
94.4.3	kernel/driver/ata.h .....	431
94.4.4	kernel/driver/ata/ata_cmd_identify_device.c .....	434
94.4.5	kernel/driver/ata/ata_cmd_read_sectors.c .....	434
94.4.6	kernel/driver/ata/ata_cmd_write_sectors.c .....	435
94.4.7	kernel/driver/ata/ata_device.c .....	436
94.4.8	kernel/driver/ata/ata_drq.c .....	437
94.4.9	kernel/driver/ata/ata_init.c .....	438
94.4.10	kernel/driver/ata/ata_lba28.c .....	441
94.4.11	kernel/driver/ata/ata_public.c .....	441
94.4.12	kernel/driver/ata/ata_rdy.c .....	442
94.4.13	kernel/driver/ata/ata_reset.c .....	442
94.4.14	kernel/driver/ata/ata_valid.c .....	443
94.4.15	kernel/driver/kbd.h .....	443
94.4.16	kernel/driver/kbd/kbd_isr.c .....	443
94.4.17	kernel/driver/kbd/kbd_load.c .....	445
94.4.18	kernel/driver/kbd/kbd_public.c .....	447
94.4.19	kernel/driver/nic/ne2k.h .....	447
94.4.20	kernel/driver/nic/ne2k/ne2k_check.c .....	449
94.4.21	kernel/driver/nic/ne2k/ne2k_isr.c .....	450
94.4.22	kernel/driver/nic/ne2k/ne2k_isr_expect.c .....	451
94.4.23	kernel/driver/nic/ne2k/ne2k_reset.c .....	452
94.4.24	kernel/driver/nic/ne2k/ne2k_rx.c .....	457
94.4.25	kernel/driver/nic/ne2k/ne2k_rx_reset.c .....	461
94.4.26	kernel/driver/nic/ne2k/ne2k_tx.c .....	462
94.4.27	kernel/driver/pci.h .....	463
94.4.28	kernel/driver/pci/pci_init.c .....	465

94.4.29	kernel/driver/pci/pci_public.c	466
94.4.30	kernel/driver/screen.h	466
94.4.31	kernel/driver/screen/screen_clear.c	467
94.4.32	kernel/driver/screen/screen_current.c	467
94.4.33	kernel/driver/screen/screen_init.c	467
94.4.34	kernel/driver/screen/screen_new_line.c	468
94.4.35	kernel/driver/screen/screen_number.c	468
94.4.36	kernel/driver/screen/screen_pointer.c	469
94.4.37	kernel/driver/screen/screen_public.c	469
94.4.38	kernel/driver/screen/screen_putc.c	469
94.4.39	kernel/driver/screen/screen_scroll.c	470
94.4.40	kernel/driver/screen/screen_select.c	470
94.4.41	kernel/driver/screen/screen_update.c	471
94.4.42	kernel/driver/tty.h	472
94.4.43	kernel/driver/tty/tty_console.c	472
94.4.44	kernel/driver/tty/tty_init.c	473
94.4.45	kernel/driver/tty/tty_public.c	474
94.4.46	kernel/driver/tty/tty_read.c	474
94.4.47	kernel/driver/tty/tty_reference.c	475
94.4.48	kernel/driver/tty/tty_write.c	475
94.5	os32: «kernel/fs.h»	475
94.5.1	kernel/fs/fd_dup.c	480
94.5.2	kernel/fs/fd_reference.c	481
94.5.3	kernel/fs/file_pipe_make.c	482
94.5.4	kernel/fs/file_reference.c	482
94.5.5	kernel/fs/file_stdio_dev_make.c	483
94.5.6	kernel/fs/fs_init.c	483
94.5.7	kernel/fs/fs_public.c	484
94.5.8	kernel/fs/inode_alloc.c	484
94.5.9	kernel/fs/inode_check.c	486
94.5.10	kernel/fs/inode_dir_empty.c	487
94.5.11	kernel/fs/inode_file_read.c	488
94.5.12	kernel/fs/inode_file_write.c	490
94.5.13	kernel/fs/inode_free.c	491
94.5.14	kernel/fs/inode_fzones_read.c	492
94.5.15	kernel/fs/inode_fzones_write.c	492
94.5.16	kernel/fs/inode_get.c	493
94.5.17	kernel/fs/inode_pipe_make.c	496
94.5.18	kernel/fs/inode_pipe_read.c	497
94.5.19	kernel/fs/inode_pipe_write.c	498
94.5.20	kernel/fs/inode_print.c	499
94.5.21	kernel/fs/inode_put.c	500
94.5.22	kernel/fs/inode_reference.c	501
94.5.23	kernel/fs/inode_save.c	503
94.5.24	kernel/fs/inode_stdio_dev_make.c	503
94.5.25	kernel/fs/inode_truncate.c	504
94.5.26	kernel/fs/inode_zone.c	506
94.5.27	kernel/fs/path_device.c	512
94.5.28	kernel/fs/path_fix.c	513
94.5.29	kernel/fs/path_full.c	514
94.5.30	kernel/fs/path_inode.c	514
94.5.31	kernel/fs/path_inode_link.c	517
94.5.32	kernel/fs/sb_inode_status.c	521
94.5.33	kernel/fs/sb_mount.c	521
94.5.34	kernel/fs/sb_print.c	524
94.5.35	kernel/fs/sb_reference.c	524
94.5.36	kernel/fs/sb_save.c	525
94.5.37	kernel/fs/sb_zone_status.c	526

94.5.38	kernel/fs/sock_free_port.c	526
94.5.39	kernel/fs/sock_reference.c	526
94.5.40	kernel/fs/zone_alloc.c	527
94.5.41	kernel/fs/zone_free.c	528
94.5.42	kernel/fs/zone_print.c	529
94.5.43	kernel/fs/zone_read.c	529
94.5.44	kernel/fs/zone_write.c	530
94.6	os32: «kernel/ibm_i386.h»	530
94.6.1	kernel/ibm_i386/_in_16.s	533
94.6.2	kernel/ibm_i386/_in_32.s	534
94.6.3	kernel/ibm_i386/_in_8.s	534
94.6.4	kernel/ibm_i386/_out_16.s	534
94.6.5	kernel/ibm_i386/_out_32.s	535
94.6.6	kernel/ibm_i386/_out_8.s	535
94.6.7	kernel/ibm_i386/cli.s	535
94.6.8	kernel/ibm_i386/gdt.c	536
94.6.9	kernel/ibm_i386/gdt_load.s	536
94.6.10	kernel/ibm_i386/gdt_print.c	537
94.6.11	kernel/ibm_i386/gdt_public.c	537
94.6.12	kernel/ibm_i386/gdt_segment.c	537
94.6.13	kernel/ibm_i386/idt.c	538
94.6.14	kernel/ibm_i386/idt_descriptor.c	539
94.6.15	kernel/ibm_i386/idt_irq_remap.c	539
94.6.16	kernel/ibm_i386/idt_load.s	540
94.6.17	kernel/ibm_i386/idt_print.c	540
94.6.18	kernel/ibm_i386/idt_public.c	541
94.6.19	kernel/ibm_i386/irq_off.c	541
94.6.20	kernel/ibm_i386/irq_on.c	541
94.6.21	kernel/ibm_i386/isr.s	542
94.6.22	kernel/ibm_i386/isr_exception_name.c	552
94.6.23	kernel/ibm_i386/isr_exception_unrecoverable.c	552
94.6.24	kernel/ibm_i386/isr_irq_clear.c	553
94.6.25	kernel/ibm_i386/isr_irq_clear_pic1.c	553
94.6.26	kernel/ibm_i386/isr_irq_clear_pic2.c	553
94.6.27	kernel/ibm_i386/sti.s	553
94.7	os32: «kernel/lib_k.h»	554
94.7.1	kernel/lib_k/k_exit.s	554
94.7.2	kernel/lib_k/k_gets.c	554
94.7.3	kernel/lib_k/k_perror.c	555
94.7.4	kernel/lib_k/k_printf.c	555
94.7.5	kernel/lib_k/k_sleep.c	555
94.7.6	kernel/lib_k/k_stime.c	556
94.7.7	kernel/lib_k/k_usleep.c	556
94.7.8	kernel/lib_k/k_vprintf.c	557
94.7.9	kernel/lib_k/k_vsprintf.c	557
94.8	os32: «kernel/lib_s.h»	558
94.8.1	kernel/lib_s/s_exit.c	560
94.8.2	kernel/lib_s/s_accept.c	562
94.8.3	kernel/lib_s/s_bind.c	564
94.8.4	kernel/lib_s/s_brk.c	565
94.8.5	kernel/lib_s/s_chdir.c	569
94.8.6	kernel/lib_s/s_chmod.c	569
94.8.7	kernel/lib_s/s_chown.c	570
94.8.8	kernel/lib_s/s_clock.c	571
94.8.9	kernel/lib_s/s_close.c	571
94.8.10	kernel/lib_s/s_connect.c	572
94.8.11	kernel/lib_s/s_dup.c	575

94.8.12	kernel/lib_s/s_dup2.c	575
94.8.13	kernel/lib_s/s_fchmod.c	575
94.8.14	kernel/lib_s/s_fchown.c	576
94.8.15	kernel/lib_s/s_fcntl.c	577
94.8.16	kernel/lib_s/s_fork.c	578
94.8.17	kernel/lib_s/s_fstat.c	582
94.8.18	kernel/lib_s/s_ipconfig.c	583
94.8.19	kernel/lib_s/s_kill.c	584
94.8.20	kernel/lib_s/s_link.c	585
94.8.21	kernel/lib_s/s_listen.c	586
94.8.22	kernel/lib_s/s_longjmp.c	587
94.8.23	kernel/lib_s/s_lseek.c	588
94.8.24	kernel/lib_s/s_mkdir.c	589
94.8.25	kernel/lib_s/s_mknod.c	591
94.8.26	kernel/lib_s/s_mount.c	592
94.8.27	kernel/lib_s/s_open.c	592
94.8.28	kernel/lib_s/s_pipe.c	596
94.8.29	kernel/lib_s/s_read.c	598
94.8.30	kernel/lib_s/s_recvfrom.c	600
94.8.31	kernel/lib_s/s_routead.c	606
94.8.32	kernel/lib_s/s_routedel.c	607
94.8.33	kernel/lib_s/s_sbrk.c	608
94.8.34	kernel/lib_s/s_send.c	609
94.8.35	kernel/lib_s/s_setegid.c	612
94.8.36	kernel/lib_s/s_setuid.c	612
94.8.37	kernel/lib_s/s_setgid.c	613
94.8.38	kernel/lib_s/s_setjmp.c	613
94.8.39	kernel/lib_s/s_setuid.c	614
94.8.40	kernel/lib_s/s_signal.c	614
94.8.41	kernel/lib_s/s_socket.c	615
94.8.42	kernel/lib_s/s_stat.c	617
94.8.43	kernel/lib_s/s_stime.c	618
94.8.44	kernel/lib_s/s_tcgetattr.c	618
94.8.45	kernel/lib_s/s_tsetattr.c	619
94.8.46	kernel/lib_s/s_time.c	620
94.8.47	kernel/lib_s/s_umount.c	620
94.8.48	kernel/lib_s/s_unlink.c	622
94.8.49	kernel/lib_s/s_wait.c	624
94.8.50	kernel/lib_s/s_write.c	625
94.9	os32: «kernel/main.h»	627
94.9.1	kernel/main/build.h	628
94.9.2	kernel/main/crt0.s	628
94.9.3	kernel/main/kmain.c	629
94.9.4	kernel/main/menu.c	634
94.9.5	kernel/main/run.c	634
94.9.6	kernel/main/stack.s	635
94.10	os32: «kernel/memory.h»	635
94.10.1	kernel/memory/mb_alloc.c	636
94.10.2	kernel/memory/mb_alloc_size.c	637
94.10.3	kernel/memory/mb_clean.c	638
94.10.4	kernel/memory/mb_free.c	638
94.10.5	kernel/memory/mb_print.c	639
94.10.6	kernel/memory/mb_public.c	640
94.10.7	kernel/memory/mb_reduce.c	640
94.10.8	kernel/memory/mb_reference.c	641
94.10.9	kernel/memory/mb_size.c	641
94.11	os32: «kernel/multiboot.h»	641

94.11.1	kernel/multiboot/mboot_cmdline_opt.c	642
94.11.2	kernel/multiboot/mboot_public.c	643
94.11.3	kernel/multiboot/mboot_save.c	643
94.12	os32: «kernel/net.h»	644
94.12.1	kernel/net/arp.h	647
94.12.2	kernel/net/arp/arp_clean.c	648
94.12.3	kernel/net/arp/arp_index.c	648
94.12.4	kernel/net/arp/arp_init.c	649
94.12.5	kernel/net/arp/arp_print.c	649
94.12.6	kernel/net/arp/arp_public.c	649
94.12.7	kernel/net/arp/arp_reference.c	649
94.12.8	kernel/net/arp/arp_request.c	650
94.12.9	kernel/net/arp/arp_rx.c	651
94.12.10	kernel/net/icmp.h	653
94.12.11	kernel/net/icmp/icmp_rx.c	653
94.12.12	kernel/net/icmp/icmp_tx.c	655
94.12.13	kernel/net/icmp/icmp_tx_echo.c	656
94.12.14	kernel/net/icmp/icmp_tx_unreachable.c	656
94.12.15	kernel/net/ip.h	657
94.12.16	kernel/net/ip/ip_checksum.c	658
94.12.17	kernel/net/ip/ip_header.c	659
94.12.18	kernel/net/ip/ip_mask.c	659
94.12.19	kernel/net/ip/ip_public.c	660
94.12.20	kernel/net/ip/ip_reference.c	660
94.12.21	kernel/net/ip/ip_rx.c	660
94.12.22	kernel/net/ip/ip_tx.c	663
94.12.23	kernel/net/net_buffer_eth.c	665
94.12.24	kernel/net/net_buffer_lo.c	665
94.12.25	kernel/net/net_eth_ip_tx.c	666
94.12.26	kernel/net/net_eth_tx.c	668
94.12.27	kernel/net/net_index.c	668
94.12.28	kernel/net/net_index_eth.c	668
94.12.29	kernel/net/net_init.c	669
94.12.30	kernel/net/net_print.c	672
94.12.31	kernel/net/net_public.c	672
94.12.32	kernel/net/net_rx.c	673
94.12.33	kernel/net/route.h	674
94.12.34	kernel/net/route/route_init.c	674
94.12.35	kernel/net/route/route_print.c	675
94.12.36	kernel/net/route/route_public.c	675
94.12.37	kernel/net/route/route_remote_to_local.c	676
94.12.38	kernel/net/route/route_remote_to_router.c	676
94.12.39	kernel/net/route/route_sort.c	677
94.12.40	kernel/net/tcp.h	679
94.12.41	kernel/net/tcp/tcp.c	679
94.12.42	kernel/net/tcp/tcp_close.c	690
94.12.43	kernel/net/tcp/tcp_connect.c	691
94.12.44	kernel/net/tcp/tcp_rx_ack.c	692
94.12.45	kernel/net/tcp/tcp_rx_data.c	693
94.12.46	kernel/net/tcp/tcp_show.c	695
94.12.47	kernel/net/tcp/tcp_status.c	695
94.12.48	kernel/net/tcp/tcp_test.c	696
94.12.49	kernel/net/tcp/tcp_tx_ack.c	697
94.12.50	kernel/net/tcp/tcp_tx_raw.c	698
94.12.51	kernel/net/tcp/tcp_tx_rst.c	699
94.12.52	kernel/net/tcp/tcp_tx_sock.c	700
94.12.53	kernel/net/udp.h	702
94.12.54	kernel/net/udp/udp_tx.c	702

94.13	os32: «kernel/part.h»	703
94.14	os32: «kernel/proc.h»	704
94.14.1	kernel/proc/proc_available.c	706
94.14.2	kernel/proc/proc_dump_memory.c	707
94.14.3	kernel/proc/proc_init.c	708
94.14.4	kernel/proc/proc_print.c	710
94.14.5	kernel/proc/proc_public.c	712
94.14.6	kernel/proc/proc_reference.c	712
94.14.7	kernel/proc/proc_sch_net.c	712
94.14.8	kernel/proc/proc_sch_signals.c	713
94.14.9	kernel/proc/proc_sch_terminals.c	714
94.14.10	kernel/proc/proc_sch_timers.c	719
94.14.11	kernel/proc/proc_scheduler.c	719
94.14.12	kernel/proc/proc_sig_chld.c	722
94.14.13	kernel/proc/proc_sig_cont.c	722
94.14.14	kernel/proc/proc_sig_core.c	723
94.14.15	kernel/proc/proc_sig_handler.c	724
94.14.16	kernel/proc/proc_sig_ignore.c	727
94.14.17	kernel/proc/proc_sig_off.c	727
94.14.18	kernel/proc/proc_sig_on.c	727
94.14.19	kernel/proc/proc_sig_status.c	727
94.14.20	kernel/proc/proc_sig_stop.c	727
94.14.21	kernel/proc/proc_sig_term.c	728
94.14.22	kernel/proc/proc_sys_exec.c	728
94.14.23	kernel/proc/proc_timer_init.c	738
94.14.24	kernel/proc/proc_wakeup_pipe_read.c	739
94.14.25	kernel/proc/proc_wakeup_pipe_write.c	739
94.14.26	kernel/proc/proc_wakeup_terminal.c	739
94.14.27	kernel/proc/ptr.c	740
94.14.28	kernel/proc/sysroutine.c	740

## 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 [ "$SUID" = 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 if [ "$R_SZ" = "$D_VM" ]
30067 then
30068     dd if=$FILE_ELF of=$FILE_OS32.text \
30069         bs=1 skip=$T_ST count=$T_SZ 2> "/dev/null"
30070     dd if=$FILE_ELF of=$FILE_OS32.rodata \
30071         bs=1 skip=$R_ST count=$R_SZ 2> "/dev/null"
30072     dd if=$FILE_ELF of=$FILE_OS32.data \
30073         bs=1 skip=$D_ST count=$D_SZ 2> "/dev/null"
30074     cat $FILE_OS32.text $FILE_OS32.rodata \
30075         $FILE_OS32.data > $FILE_OS32
30076     #
30077     #rm $FILE_OS32.text
30078     #rm $FILE_OS32.rodata
30079     #rm $FILE_OS32.data
30080     #
30081     chmod a+x $FILE_OS32
30082 elif [ "$R_SZ" -lt "$D_VM" ]
30083 then
30084     dd if=$FILE_ELF of=$FILE_OS32.text \
30085         bs=1 skip=$T_ST count=$T_SZ 2> "/dev/null"
30086     dd if=$FILE_ELF of=$FILE_OS32.rodata \
30087         bs=1 skip=$R_ST count=$R_SZ 2> "/dev/null"
30088     dd if=/dev/zero of=$FILE_OS32.rodata-space \
30089         bs=1 count=$((($D_VM-$R_SZ)) 2> "/dev/null"
30090     dd if=$FILE_ELF of=$FILE_OS32.data \
30091         bs=1 skip=$D_ST count=$D_SZ 2> "/dev/null"
30092     #
30093     cat $FILE_OS32.text \
30094         $FILE_OS32.rodata \
30095         $FILE_OS32.rodata-space \
30096         $FILE_OS32.data > $FILE_OS32
30097     #
30098     #rm $FILE_OS32.text
30099     #rm $FILE_OS32.rodata
30100     #rm $FILE_OS32.rodata-space
30101     #rm $FILE_OS32.data
30102     #
30103     chmod a+x $FILE_OS32
30104 else
30105     echo "[${0}] ERROR: $FILE_ELF has DATA section"
30106     echo "[${0}]     not contiguous:"
30107     echo "[${0}]     RODATA end at $R_SZ, and DATA"
30108     echo "[${0}]     should start at $D_VM!"
30109 fi
30110

```

#### 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='fdisk -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='fdisk -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 -ndefaultlibs" \
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 ..... 417

94.2.2 kernel/blk/blk\_cache\_check.c ..... 418

94.2.3 kernel/blk/blk\_cache\_init.c ..... 419

94.2.4 kernel/blk/blk\_cache\_read.c ..... 419

94.2.5 kernel/blk/blk\_cache\_save.c ..... 419

94.2.6 kernel/blk/blk\_public.c ..... 420

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	421
94.3.2	kernel/dev/dev_dm.c	422
94.3.3	kernel/dev/dev_io.c	423
94.3.4	kernel/dev/dev_kmem.c	423
94.3.5	kernel/dev/dev_mem.c	426
94.3.6	kernel/dev/dev_tty.c	427

#### 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 // Everything was right, so 'n' is valid for both
200104 // read or write.
200105 //
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	430
94.4.2	kernel/dm/dm_public.c	431
94.4.3	kernel/driver/ata.h	431
94.4.4	kernel/driver/ata/ata_cmd_identify_device.c	434
94.4.5	kernel/driver/ata/ata_cmd_read_sectors.c	434
94.4.6	kernel/driver/ata/ata_cmd_write_sectors.c	435
94.4.7	kernel/driver/ata/ata_device.c	436
94.4.8	kernel/driver/ata/ata_drq.c	437
94.4.9	kernel/driver/ata/ata_init.c	438
94.4.10	kernel/driver/ata/ata_lba28.c	441
94.4.11	kernel/driver/ata/ata_public.c	441
94.4.12	kernel/driver/ata/ata_rdy.c	442
94.4.13	kernel/driver/ata/ata_reset.c	442
94.4.14	kernel/driver/ata/ata_valid.c	443
94.4.15	kernel/driver/kbd.h	443
94.4.16	kernel/driver/kbd/kbd_isr.c	443
94.4.17	kernel/driver/kbd/kbd_load.c	445
94.4.18	kernel/driver/kbd/kbd_public.c	447
94.4.19	kernel/driver/nic/ne2k.h	447
94.4.20	kernel/driver/nic/ne2k/ne2k_check.c	449
94.4.21	kernel/driver/nic/ne2k/ne2k_isr.c	450
94.4.22	kernel/driver/nic/ne2k/ne2k_isr_expect.c	451

94.4.23	kernel/driver/nic/ne2k/ne2k_reset.c	452
94.4.24	kernel/driver/nic/ne2k/ne2k_rx.c	457
94.4.25	kernel/driver/nic/ne2k/ne2k_rx_reset.c	461
94.4.26	kernel/driver/nic/ne2k/ne2k_tx.c	462
94.4.27	kernel/driver/pci.h	463
94.4.28	kernel/driver/pci/pci_init.c	465
94.4.29	kernel/driver/pci/pci_public.c	466
94.4.30	kernel/driver/screen.h	466
94.4.31	kernel/driver/screen/screen_clear.c	467
94.4.32	kernel/driver/screen/screen_current.c	467
94.4.33	kernel/driver/screen/screen_init.c	467
94.4.34	kernel/driver/screen/screen_new_line.c	468
94.4.35	kernel/driver/screen/screen_number.c	468
94.4.36	kernel/driver/screen/screen_pointer.c	469
94.4.37	kernel/driver/screen/screen_public.c	469
94.4.38	kernel/driver/screen/screen_putc.c	469
94.4.39	kernel/driver/screen/screen_scroll.c	470
94.4.40	kernel/driver/screen/screen_select.c	470
94.4.41	kernel/driver/screen/screen_update.c	471
94.4.42	kernel/driver/tty.h	472
94.4.43	kernel/driver/tty/tty_console.c	472
94.4.44	kernel/driver/tty/tty_init.c	473
94.4.45	kernel/driver/tty/tty_public.c	474
94.4.46	kernel/driver/tty/tty_read.c	474
94.4.47	kernel/driver/tty/tty_reference.c	475
94.4.48	kernel/driver/tty/tty_write.c	475

#### 94.4.1 kernel/dm/dm\_init.c

« Si veda la sezione 93.5.

```

270001 #include <kernel/dm.h>
270002 #include <kernel/part.h>
270003 #include <kernel/driver/ata.h>
270004 #include <kernel/lib_k.h>
270005 #include <stdint.h>
270006 #include <errno.h>
270007 //-----
270008 void
270009 dm_init (void)
270010 {
270011     int d;
270012     int a;
270013     int p;
270014     ata_sector_t sector_buffer;
270015     part_t *part;
270016     int status;
270017     //
270018     // Reset the data-memory table.
270019     //
270020     for (d = 0; d < DM_MAX_DEVICES; d++)
270021     {
270022         dm_table[d].type = DM_TYPE_NONE;
270023         dm_table[d].table = NULL;
270024         dm_table[d].part[0].start = 0;
270025         dm_table[d].part[0].size = 0;
270026         dm_table[d].part[0].type = PART_TYPE_NO_PART;
270027         for (p = 0; p < PART_MAX; p++)
270028         {
270029             dm_table[d].part[p + 1].start = 0;
270030             dm_table[d].part[p + 1].size = 0;
270031             dm_table[d].part[p + 1].type = PART_TYPE_NONE;
270032         }
270033     }
270034     //
270035     // Reset data-memory index.
270036     //
270037     d = 0;
270038     //
270039     // Init ATA devices.

```

```

270040 //
270041 ata_init ();
270042 //
270043 // Assign ATA devices to the first data-memory
270044 // items.
270045 //
270046 for (a = 0; a < ATA_MAX_DEVICES; a++)
270047 {
270048     if (ata_table[a].present == 0)
270049     {
270050         //
270051         // Current data-memory device will be
270052         // used for the next ATA device, if any.
270053         //
270054         continue;
270055     }
270056     //
270057     // Show something.
270058     //
270059     k_printf ("[%s] ATA drive=%i total sectors=%i\n",
270060             __func__, a, (int) ata_table[a].sectors);
270061     //
270062     dm_table[d].type = DM_TYPE_ATA;
270063     dm_table[d].table = &ata_table[a];
270064     dm_table[d].part[0].start = 0;
270065     dm_table[d].part[0].size = ata_table[a].sectors;
270066     dm_table[d].part[0].type = PART_TYPE_NO_PART;
270067     //
270068     // Read partitions.
270069     //
270070     status = ata_read_sector (a, 0, &sector_buffer);
270071     //
270072     if (status)
270073     {
270074         errset (errno);
270075         k_perror (NULL);
270076     }
270077     else
270078     {
270079         part =
270080             ((void *) &sector_buffer) + PART_TABLE_OFF;
270081         //
270082         for (p = 0; p < PART_MAX; p++)
270083         {
270084             //
270085             dm_table[d].part[p + 1].start =
270086                 part->l_start * ATA_SECTOR_SIZE;
270087             dm_table[d].part[p + 1].size =
270088                 part->size * ATA_SECTOR_SIZE;
270089             dm_table[d].part[p + 1].type = part->type;
270090             //
270091             // Show info.
270092             //
270093             if (part->type != 0)
270094             {
270095                 k_printf ("[%s] partition type=%02x "
270096                         "start sector=%i "
270097                         "total sectors=%i\n",
270098                         __func__, (int) part->type,
270099                         (int) part->l_start,
270100                         (int) part->size);
270101             }
270102             //
270103             part++;
270104         }
270105     }
270106     //
270107     // Next data-memory device.
270108     //
270109     d++;
270110 }
270111 }

```

#### 94.4.2 kernel/dm/dm\_public.c

« Si veda la sezione 93.5.

```

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

```

#### 94.4.3 kernel/driver/ata.h

« Si veda la sezione 93.2.

```

290001 #ifndef _KERNEL_DRIVER_ATA_H
290002 #define _KERNEL_DRIVER_ATA_H 1

```



```

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_altername = 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_altername = 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_altername = 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_altername = 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_altername = 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_altername = 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] = '1';
430024     kbd.map1[3] = '2';
430025     kbd.map2[3] = '2';
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] = '^';    // i, ^
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] = '{';    // ò, 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, ¢
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_CLDAL   0x02 // r-
450019 #define NE2K_PSTOP   0x02 // -w
450020 #define NE2K_BNRY    0x03 // iw
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 // iw
450028 #define NE2K_CRDA0  0x08 // r-
450029 #define NE2K_RSAR0  0x08 // -w
450030 #define NE2K_CRDAL  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 // iw
450046 #define NE2K_PAR1    0x02 // iw
450047 #define NE2K_PAR2    0x03 // iw
450048 #define NE2K_PAR3    0x04 // iw
450049 #define NE2K_PAR4    0x05 // iw
450050 #define NE2K_PAR5    0x06 // iw
450051 #define NE2K_CURR    0x07 // iw
450052 #define NE2K_MAR0    0x08 // iw
450053 #define NE2K_MAR1    0x09 // iw
450054 #define NE2K_MAR2    0x0A // iw
450055 #define NE2K_MAR3    0x0B // iw
450056 #define NE2K_MAR4    0x0C // iw
450057 #define NE2K_MAR5    0x0D // iw
450058 #define NE2K_MAR6    0x0E // iw
450059 #define NE2K_MAR7    0x0F // iw
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 //      | / |

```

```

451001 //      | / |
451002 //      | / |
451003 //      | / |
451004 //      | / |
451005 //      | / |
451006 // [1] 0x600 is equal to 1536, that is the max space
451007 //      that a packet can occupy inside the ring buffer.
451008 //      So, there is the place for a
451009 //      single transmission packet.
451010 //
451011 // Local DMA addresses:
451012 //
451013 #define NE2K_TX_START 0x40 // Means: 0x4000
451014 #define NE2K_TX_STOP 0x46 // Means: 0x4600
451015 //
451016 #define NE2K_RX_START 0x46 // Means: 0x4600
451017 #define NE2K_RX_STOP 0xC0 // Means: 0xC000
451018 //
451019 // Transmit buffer (remote DMA).
451020 //
451021 #define NE2K_TX_BUFFER NE2K_TX_START
451022 //
451023 // SA-PROM (Station Address PROM) size
451024 //
451025 #define NE2K_SAPROM_SIZE 32 // 32 bytes
451026 //-----|
451027 int ne2k_check (uintptr_t io);
451028 int ne2k_isr (uintptr_t io);
451029 int ne2k_isr_expect (uintptr_t io, unsigned int isr_expect);
451030 int ne2k_reset (uintptr_t io, void *address);
451031 int ne2k_rx (uintptr_t io);
451032 int ne2k_rx_reset (uintptr_t io);
451033 int ne2k_tx (uintptr_t io, void *buffer, size_t size);
451034 //
451035 void ne2k_test2 (int eth);
451036 //
451037 //-----|
451038 //
451039 #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_0d);
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 | 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 | 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 | 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 | 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 // | -- |RDC|CNTE|OVWE|TXE|RXE|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 bnrty = 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 bnrty.
500103 //
500104 out_8 ((io + NE2K_RSAR0), 0x00); // Must be
500105 // zero.
500106 out_8 ((io + NE2K_RSAR1), bnrty);
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|OVN|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, bnrty, curr, NE2K_RX_STOP,
500169      frame_size);
500170 }
500171 //
500172 if (next == bnrty)
500173 {
500174     k_printf
500175     ("[%s] next==bnrty but should "
500176      "not happen!\n", __func__);
500177     errset (E_DRIVER_FAULT);
500178     return (-1);
500179 }
500180 //
500181 if (next > bnrty)
500182 {
500183     bytes = frame_size;
500184 }
500185 //
500186 if (next < bnrty)
500187 {
500188     //
500189     // Read up to the bottom.
500190     //
500191     bytes = ((NE2K_RX_STOP - bnrty) * 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), bnrty);
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 | 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 | 0 | 1 | 0 | 1 | 0 | 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 | 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 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 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 | 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 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 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 & 0xFFFFF000;
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 //
750108 // Please note that the type 'sb_t' is declared
750109 // before the type 'inode_t', but this member
750110 // 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     inode_t ino; // Inode number.
750144     sb_t *sb_attached; // [8]
750145     blkcnt_t blkcnt; // Rounded size/blksize.
750146     unsigned char references; // Run time active
750147 // references.
750148     char changed:1, // 1 == to be saved.
750149     pipe_dir:1; // 0 == read, 1 == write.
750150     unsigned char pipe_off_read; // Pipe read offset.
750151     unsigned char pipe_off_write; // Pipe write offset
750152     unsigned char pipe_ref_read; // Pipe read
750153 // references.
750154     unsigned char pipe_ref_write; // Pipe write
750155 // references
750156 };
750157
750158 extern inode_t inode_table[INODE_MAX_SLOTS];
750159 //
750160 // [7] the member 'sb' must be kept at the same
750161 // position, because it is used to calculate the
750162 // inode header size, saved on disk.
750163 //
750164 // [8] If the inode is a mount point for another
750165 // device, the other super block pointer is saved
750166 // inside 'sb_attached'.
750167 //
750168 -----
750169 #define SOCK_MAX_SLOTS 64
750170 #define SOCK_MAX_QUEUE (SOCK_MAX_SLOTS/4)
750171 //
750172 struct sock
750173 {
750174     int family;
750175     int type;
750176     int protocol;
750177     h_addr_t laddr; // Local address, host byte
750178 // order.
750179     h_port_t lport; // Local port, host byte
750180 // order.
750181     h_addr_t raddr; // Remote address, host byte
750182 // order.
750183     h_port_t rport; // Remote port, host byte
750184 // order.
750185     struct
750186     {
750187         clock_t clock[IP_MAX_PACKETS]; // [9]
750188     } read;
750189     uint8_t active:1, // Is the socket used?
750190     unreachable:1, //
750191     unreachable_host:1, // ICMP unreachable status.
750192     unreachable_prot:1, //
750193     unreachable_port:1; //
750194     struct
750195     {
750196         uint16_t conn:4, // Connection status.

```

```

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

```

```

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

```

94.5.1 kernel/fs/fd\_dup.c .....480

94.5.2 kernel/fs/fd\_reference.c .....481

94.5.3	kernel/fs/file_pipe_make.c	482
94.5.4	kernel/fs/file_reference.c	482
94.5.5	kernel/fs/file_stdio_dev_make.c	483
94.5.6	kernel/fs/fs_init.c	483
94.5.7	kernel/fs/fs_public.c	484
94.5.8	kernel/fs/inode_alloc.c	484
94.5.9	kernel/fs/inode_check.c	486
94.5.10	kernel/fs/inode_dir_empty.c	487
94.5.11	kernel/fs/inode_file_read.c	488
94.5.12	kernel/fs/inode_file_write.c	490
94.5.13	kernel/fs/inode_free.c	491
94.5.14	kernel/fs/inode_fzones_read.c	492
94.5.15	kernel/fs/inode_fzones_write.c	492
94.5.16	kernel/fs/inode_get.c	493
94.5.17	kernel/fs/inode_pipe_make.c	496
94.5.18	kernel/fs/inode_pipe_read.c	497
94.5.19	kernel/fs/inode_pipe_write.c	498
94.5.20	kernel/fs/inode_print.c	499
94.5.21	kernel/fs/inode_put.c	500
94.5.22	kernel/fs/inode_reference.c	501
94.5.23	kernel/fs/inode_save.c	503
94.5.24	kernel/fs/inode_stdio_dev_make.c	503
94.5.25	kernel/fs/inode_truncate.c	504
94.5.26	kernel/fs/inode_zone.c	506
94.5.27	kernel/fs/path_device.c	512
94.5.28	kernel/fs/path_fix.c	513
94.5.29	kernel/fs/path_full.c	514
94.5.30	kernel/fs/path_inode.c	514
94.5.31	kernel/fs/path_inode_link.c	517
94.5.32	kernel/fs/sb_inode_status.c	521
94.5.33	kernel/fs/sb_mount.c	521
94.5.34	kernel/fs/sb_print.c	524
94.5.35	kernel/fs/sb_reference.c	524
94.5.36	kernel/fs/sb_save.c	525
94.5.37	kernel/fs/sb_zone_status.c	526
94.5.38	kernel/fs/sock_free_port.c	526
94.5.39	kernel/fs/sock_reference.c	526
94.5.40	kernel/fs/zone_alloc.c	527
94.5.41	kernel/fs/zone_free.c	528
94.5.42	kernel/fs/zone_print.c	529
94.5.43	kernel/fs/zone_read.c	529
94.5.44	kernel/fs/zone_write.c	530

#### 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

&lt;

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

&gt;

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             strcmp (dir->name, ".", NAME_MAX) != 0
850046                 && strcmp (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     k_printf
860074         ("inode_fzones_read (inode, fzone %i,... )\n",
860075          fzone);
860076     errset (EUNKNOWN);
860077     return (0);    // Zero bytes read!
860078 }
860079 //
860080 //
860081 // The first file-zone was read: copy it inside the
860082 // destination
860083 // buffer and continue reading the other zones
860084 // needed. Variables
860085 // 'off_buffer' (destination buffer index) and
860086 // 'size_read' (copy
860087 // byte counter) must be reset here. Variable
860088 // 'off_fzone' is already
860089 // set with the initial offset inside 'zone_buffer'.
860090 //
860091 off_buffer = 0;
860092 size_read = 0;
860093 //
860094 while (count)
860095 {
860096     //
860097     // Copy the zone buffer into the destination.
860098     // Variables
860099     // 'off_fzone', 'off_buffer' and 'size_read'
860100     // must not be
860101     // initialized inside the loop.
860102     //
860103     for (;
860104          off_fzone < inode->sb->blksize && count > 0;
860105          off_fzone++, off_buffer++, size_read++,
860106          count--, offset++)
860107     {
860108         destination[off_buffer] = zone_buffer[off_fzone];
860109     }
860110     //
860111     // If not all the bytes are copied, read the
860112     // next file-zone.
860113     //
860114     if (count)
860115     {
860116         //
860117         // Read another file-zone inside the zone
860118         // buffer.
860119         // Again, the function 'inode_fzones_read()'
860120         // might
860121         // return a null pointer, but the variable
860122         // 'errno' tells if
860123         // it is really an error. For this reason,
860124         // the variable
860125         // 'errno' must be reset before the read,
860126         // and checked after
860127         // it.
860128         //
860129         fzone = offset / inode->sb->blksize;
860130         off_fzone = offset % inode->sb->blksize;
860131         blkcnt_read =
860132             inode_fzones_read (inode, fzone,
860133                               zone_buffer, (blkcnt_t) 1);
860134         if (blkcnt_read <= 0)
860135         {

```

```

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.

```

100001 #include <kernel/fs.h>
100002 #include <errno.h>
100003 #include <kernel/lib_k.h>
100004 //-----
100005 int
100006 inode_truncate (inode_t * inode)
100007 {
100008     unsigned int indirect_zones;
100009     zno_t zone_table1[INODE_MAX_INDIRECT_ZONES];
100010     zno_t zone_table2[INODE_MAX_INDIRECT_ZONES];
100011     unsigned int i; // Direct index.
100012     unsigned int i0; // Single indirect index.
100013     unsigned int i1; // Double indirect first
100014     // index.
100015     unsigned int i2; // Double indirect second
100016     // index.
100017     int status; // 'zone_read()' return value.
100018     //
100019     // Check argument.
100020     //

```

```

100021     if (inode == NULL)
100022     {
100023         errset (EINVAL);
100024         return (-1);
100025     }
100026     //
100027     // Calculate how many indirect zone numbers are
100028     // stored inside
100029     // a zone: it depends on the zone size.
100030     //
100031     indirect_zones = inode->sb->blksize / 2;
100032     //
100033     // Scan and release direct zones. Errors are
100034     // ignored.
100035     //
100036     for (i = 0; i < 7; i++)
100037     {
100038         zone_free (inode->sb, inode->direct[i]);
100039         inode->direct[i] = 0;
100040     }
100041     //
100042     // Scan single indirect zones, if present.
100043     //
100044     if (inode->blkcnt > 7 && inode->indirect1 != 0)
100045     {
100046         //
100047         // There is a single indirect table to load.
100048         // Errors are
100049         // almost ignored.
100050         //
100051         status =
100052             zone_read (inode->sb, inode->indirect1,
100053                       zone_table1);
100054         if (status == 0)
100055         {
100056             //
100057             // Scan the table and remove zones.
100058             //
100059             for (i0 = 0; i0 < indirect_zones; i0++)
100060             {
100061                 zone_free (inode->sb, zone_table1[i0]);
100062             }
100063         }
100064         //
100065         // Remove indirect table too.
100066         //
100067         zone_free (inode->sb, inode->indirect1);
100068         //
100069         // Clear single indirect reference inside the
100070         // inode.
100071         //
100072         inode->indirect1 = 0;
100073     }
100074     //
100075     // Scan double indirect zones, if present.
100076     //
100077     if (inode->blkcnt > (7 + indirect_zones)
100078         && inode->indirect2 != 0)
100079     {
100080         //
100081         // There is a double indirect table to load.
100082         // Errors are
100083         // almost ignored.
100084         //
100085         status =
100086             zone_read (inode->sb, inode->indirect2,
100087                       zone_table1);
100088         if (status == 0)
100089         {
100090             //
100091             // Scan the table and get second level
100092             // indirection.
100093             //
100094             for (i1 = 0; i1 < indirect_zones; i1++)
100095             {
100096                 if ((inode->blkcnt
100097                     >
100098                     (7 + indirect_zones +
100099                     indirect_zones * i1))
100100                     && zone_table1[i1] != 0)
100101                 {
100102                     //
100103                     // There is a second level table to
100104                     // load.
100105                     //
100106                     status =
100107                         zone_read (inode->sb,

```

```

100008         zone_table1[i1],
100009         zone_table2);
100010     if (status == 0)
100011     {
100012         //
100013         // Release zones.
100014         //
100015         for (i2 = 0;
100016             i2 < indirect_zones &&
100017             (inode->blkcnt >
100018              (7 + indirect_zones +
100019               indirect_zones * i1 +
100020                i2)); i2++)
100021         {
100022             zone_free (inode->sb,
100023                       zone_table2[i2]);
100024         }
100025         //
100026         // Remove second level indirect
100027         // table.
100028         //
100029         zone_free (inode->sb,
100030                   zone_table1[i1]);
100031     }
100032 }
100033 //
100034 // Remove first level indirect table.
100035 //
100036 //
100037 zone_free (inode->sb, inode->indirect2);
100038 }
100039 //
100040 // Clear single indirect reference inside the
100041 // inode.
100042 //
100043 inode->indirect2 = 0;
100044 }
100045 //
100046 // Update super block and inode data.
100047 //
100048 sb_save (inode->sb);
100049 inode->size = 0;
100050 inode->changed = 1;
100051 inode_save (inode);
100052 //
100053 // Successful return.
100054 //
100055 return (0);
100056 }

```

#### 94.5.26 kernel/fs/inode\_zone.c

« Si veda la sezione 93.6.25.

```

100001 #include <kernel/fs.h>
100002 #include <errno.h>
100003 #include <kernel/lib_k.h>
100004 //-----
100005 zno_t
100006 inode_zone (inode_t * inode, zno_t fzone, int write)
100007 {
100008     unsigned int indirect_zones;
100009     unsigned int allocated_zone;
100010     zno_t zone_table[INODE_MAX_INDIRECT_ZONES];
100011     char buffer[SB_MAX_ZONE_SIZE];
100012     unsigned int i0; // Single indirect index.
100013     unsigned int i1; // Double indirect first
100014     // index.
100015     unsigned int i2; // Double indirect second
100016     // index.
100017     int status;
100018     zno_t zone_second; // Second level table zone.
100019     //
100020     // Check to have a valid inode.
100021     //
100022     if (inode == NULL)
100023     {
100024         errset (EINVAL);
100025         return ((zno_t) - 1);
100026     }
100027     //
100028     // Calculate how many indirect zone numbers are
100029     // stored inside
100030     // a zone: it depends on the zone size.
100031     //
100032     indirect_zones = inode->sb->blksize / 2;
100033     //

```

```

100034 // Convert file-zone number into a zone number.
100035 //
100036 if (fzone < 7)
100037 {
100038     //
100039     // 0 <= fzone <= 6
100040     // The zone number is inside the direct zone
100041     // references.
100042     // Verify to have such zone.
100043     //
100044     if (inode->direct[fzone] == 0)
100045     {
100046         //
100047         // There is not such zone, but we do not
100048         // consider
100049         // it an error, because a file can be not
100050         // contiguous.
100051         //
100052         if (!write)
100053         {
100054             return ((zno_t) 0);
100055         }
100056         //
100057         // Must be allocated.
100058         //
100059         allocated_zone = zone_alloc (inode->sb);
100060         if (allocated_zone == 0)
100061         {
100062             //
100063             // Cannot allocate the zone. The
100064             // variable 'errno' is
100065             // set by 'zone_alloc()'.
100066             //
100067             return ((zno_t) - 1);
100068         }
100069         //
100070         // The zone is allocated: clear the zone and
100071         // save.
100072         //
100073         memset (buffer, 0, SB_MAX_ZONE_SIZE);
100074         status =
100075             zone_write (inode->sb, allocated_zone, buffer);
100076         if (status < 0)
100077         {
100078             //
100079             // Cannot overwrite the zone. The
100080             // variable 'errno' is
100081             // set by 'zone_write()'.
100082             //
100083             return ((zno_t) - 1);
100084         }
100085         //
100086         // The zone is allocated and cleared: save
100087         // the inode.
100088         //
100089         inode->direct[fzone] = allocated_zone;
100090         inode->changed = 1;
100091         status = inode_save (inode);
100092         if (status != 0)
100093         {
100094             //
100095             // Cannot save the inode. The variable
100096             // 'errno' is
100097             // set 'inode_save()'.
100098             //
100099             return ((zno_t) - 1);
100100         }
100101     }
100102     //
100103     // The zone is there: return it.
100104     //
100105     return (inode->direct[fzone]);
100106 }
100107 if (fzone < 7 + indirect_zones)
100108 {
100109     //
100110     // 7 <= fzone <= (6 + indirect_zones)
100111     // The zone number is inside the single indirect
100112     // zone
100113     // references: verify to have the indirect zone
100114     // table.
100115     //
100116     if (inode->indirect1 == 0)
100117     {
100118         //
100119         // There is not such zone, but it is not an
100120         // error.

```



```

100121 //
100122 if (!write)
100123 {
100124     return ((zno_t) 0);
100125 }
100126 //
100127 // The first level of indirection must be
100128 // initialized.
100129 //
100130 allocated_zone = zone_alloc (inode->sb);
100131 if (allocated_zone == 0)
100132 {
100133     //
100134     // Cannot allocate the zone for the
100135     // indirection table:
100136     // this is an error and the 'errno'
100137     // value is produced
100138     // by 'zone_alloc()'.
100139     //
100140     return ((zno_t) - 1);
100141 }
100142 //
100143 // The zone for the indirection table is
100144 // allocated:
100145 // clear the zone and save.
100146 //
100147 memset (buffer, 0, SB_MAX_ZONE_SIZE);
100148 status =
100149     zone_write (inode->sb, allocated_zone, buffer);
100150 if (status < 0)
100151 {
100152     //
100153     // Cannot overwrite the zone. The
100154     // variable 'errno' is
100155     // set by 'zone_write()'.
100156     //
100157     return ((zno_t) - 1);
100158 }
100159 //
100160 // The indirection table zone is allocated
100161 // and cleared:
100162 // save the inode.
100163 //
100164 inode->indirect1 = allocated_zone;
100165 inode->changed = 1;
100166 status = inode_save (inode);
100167 if (status != 0)
100168 {
100169     //
100170     // Cannot save the inode. This is an
100171     // error and the value
100172     // for 'errno' is produced by
100173     // 'inode_save()'.
100174     //
100175     return ((zno_t) - 1);
100176 }
100177 }
100178 //
100179 // An indirect table is present inside the file
100180 // system:
100181 // load it.
100182 //
100183 status =
100184     zone_read (inode->sb, inode->indirect1, zone_table);
100185 if (status != 0)
100186 {
100187     //
100188     // Cannot load the indirect table. This is
100189     // an error and the
100190     // value for 'errno' is assigned by function
100191     // 'zone_read()'.
100192     //
100193     return ((zno_t) - 1);
100194 }
100195 //
100196 // The indirect table was read. Calculate the
100197 // index inside
100198 // the table, for the requested zone.
100199 //
100200 i0 = (fzone - 7);
100201 //
100202 // Check if the zone is to be allocated.
100203 //
100204 if (zone_table[i0] == 0)
100205 {
100206     //
100207     // There is not such zone, but it is not an

```

```

100208 // error.
100209 //
100210 if (!write)
100211 {
100212     return ((zno_t) 0);
100213 }
100214 //
100215 // The zone must be allocated.
100216 //
100217 allocated_zone = zone_alloc (inode->sb);
100218 if (allocated_zone == 0)
100219 {
100220     //
100221     // There is no space for the zone
100222     // allocation. The
100223     // variable 'errno' is already updated
100224     // by
100225     // 'zone_alloc()'.
100226     //
100227     return ((zno_t) - 1);
100228 }
100229 //
100230 // The zone is allocated: clear the zone and
100231 // save.
100232 //
100233 memset (buffer, 0, SB_MAX_ZONE_SIZE);
100234 status =
100235     zone_write (inode->sb, allocated_zone, buffer);
100236 if (status < 0)
100237 {
100238     //
100239     // Cannot overwrite the zone. The
100240     // variable 'errno' is
100241     // set by 'zone_write()'.
100242     //
100243     return ((zno_t) - 1);
100244 }
100245 //
100246 // The zone is allocated and cleared: update
100247 // the indirect
100248 // zone table and save it. The inode is not
100249 // modified,
100250 // because the indirect table is outside.
100251 //
100252 zone_table[i0] = allocated_zone;
100253 status =
100254     zone_write (inode->sb, inode->indirect1,
100255                 zone_table);
100256 if (status != 0)
100257 {
100258     //
100259     // Cannot save the zone. The variable
100260     // 'errno' is already
100261     // set by 'zone_write()'.
100262     //
100263     return ((zno_t) - 1);
100264 }
100265 }
100266 //
100267 // The zone is allocated.
100268 //
100269 return (zone_table[i0]);
100270 }
100271 else
100272 {
100273     //
100274     // (7 + indirect_zones) <= fzone
100275     // The zone number is inside the double indirect
100276     // zone
100277     // references.
100278     // Verify to have the first level of second
100279     // indirection.
100280     //
100281     if (inode->indirect2 == 0)
100282     {
100283         //
100284         // There is not such zone, but it is not an
100285         // error.
100286         //
100287         if (!write)
100288         {
100289             return ((zno_t) 0);
100290         }
100291         //
100292         // The first level of second indirection
100293         // must be
100294         // 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     //
1050111     // The size read must be a multiple of 16.
1050112     //
1050113     size_read = ((size_read / 16) * 16);
1050114     //
1050115     // Check anyway if it is zero.
1050116     //
1050117     if (size_read == 0)
1050118     {
1050119         //
1050120         // The directory is ended: release the
1050121         // inode and return.
1050122         //
1050123         inode_put (inode);
1050124         errset (ENOENT); // No such file or
1050125         // directory.
1050126         return (NULL);
1050127     }
1050128     //
1050129     // Calculate how many directory items we
1050130     // have read.
1050131     //
1050132     dir_size_read = size_read / 16;
1050133     //
1050134     // Scan the directory to find the current
1050135     // name.
1050136     //
1050137     for (d = 0; d < dir_size_read; d++)
1050138     {
1050139         //
1050140         // Ensure to have a null terminated
1050141         // string for
1050142         // the name found.
1050143         //
1050144         memcpy (dir_name, dir[d].name,
1050145                 (size_t) NAME_MAX);
1050146         dir_name[NAME_MAX] = 0;
1050147         //
1050148         comp = strcmp (name, dir_name);
1050149         if (comp == 0 && dir[d].ino != 0)
1050150         {
1050151             //
1050152             // Found the name and verified that
1050153             // it has a link to
1050154             // a inode. Now release the
1050155             // directory inode.
1050156             //
1050157             inode_put (inode);
1050158             //
1050159             // Get next inode and break the
1050160             // loop.
1050161             //
1050162             inode = inode_get (device, dir[d].ino);
1050163             break;
1050164         }
1050165     }
1050166     //
1050167     // If index 'd' is in a valid range, the
1050168     // name was found.
1050169     //
1050170     if (d < dir_size_read)
1050171     {
1050172         //
1050173         // The name was found.
1050174         //
1050175         break;
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     //

```

```

100028 if (path == NULL || strlen (path) == 0)
100029 {
100030     errset (EINVAL); // Invalid argument:
100031     return (NULL); // the path is mandatory.
100032 }
100033 //
100034 if (inode == NULL && mode == 0)
100035 {
100036     errset (EINVAL); // Invalid argument: if the
100037     // inode is to
100038     return (NULL); // be created, the mode is
100039     // mandatory.
100040 }
100041 //
100042 if (inode != NULL)
100043 {
100044     if (mode != 0)
100045     {
100046         errset (EINVAL); // Invalid argument:
100047         // if the inode is
100048         return (NULL); // already present,
100049         // the creation mode
100050     } // must not be given.
100051     if (S_ISDIR (inode->mode))
100052     {
100053         errset (EPERM); // Operation not
100054         // permitted.
100055         return (NULL); // Refuse to link
100056         // directory.
100057     }
100058     if (inode->links >= LINK_MAX)
100059     {
100060         errset (EMLINK); // Too many links.
100061         return (NULL);
100062     }
100063 }
100064 //
100065 // Get process.
100066 //
100067 ps = proc_reference (pid);
100068 //
100069 // If the destination path already exists, the link
100070 // cannot be made.
100071 // It does not matter if the inode is known or not.
100072 //
100073 inode_new = path_inode ((uid_t) 0, path);
100074 if (inode_new != NULL)
100075 {
100076     //
100077     // A file already exists with the same name.
100078     //
100079     inode_put (inode_new);
100080     errset (EEXIST); // File exists.
100081     return (NULL);
100082 }
100083 //
100084 // At this point, 'inode_new' is 'NULL'.
100085 // Copy the source path inside the directory path
100086 // and name arrays.
100087 //
100088 strncpy (path_copy1, path, PATH_MAX);
100089 strncpy (path_copy2, path, PATH_MAX);
100090 //
100091 // Reduce to directory name and find the last name.
100092 //
100093 path_directory = dirname (path_copy1);
100094 path_name = basename (path_copy2);
100095 if (strlen (path_directory) == 0
100096     || strlen (path_name) == 0)
100097 {
100098     errset (EACCES); // Permission denied: maybe
100099     // the
100100     // original path is the root directory
100101     // and cannot find a previous directory.
100102     return (NULL);
100103 }
100104 //
100105 // Get the directory inode.
100106 //
100107 inode_directory = path_inode (pid, path_directory);
100108 if (inode_directory == NULL)
100109 {
100110     errset (errno);
100111     return (NULL);
100112 }
100113 //
100114 // Check if something is mounted on it.

```

```

100015 //
100016 if (inode_directory->sb_attached != NULL)
100017 {
100018     //
100019     // Must select the right directory.
100020     //
100021     device = inode_directory->sb_attached->device;
100022     inode_put (inode_directory);
100023     inode_directory = inode_get (device, 1);
100024     if (inode_directory == NULL)
100025     {
100026         return (NULL);
100027     }
100028 }
100029 //
100030 // If the inode to link is known, check if the
100031 // selected directory
100032 // has the same super block than the inode to link.
100033 //
100034 if (inode != NULL && inode_directory->sb != inode->sb)
100035 {
100036     inode_put (inode_directory);
100037     errset (ENOENT); // No such file or directory.
100038     return (NULL);
100039 }
100040 //
100041 // Check if write is allowed for the file system.
100042 //
100043 if (inode_directory->sb->options & MOUNT_RO)
100044 {
100045     inode_put (inode_directory);
100046     errset (EROFS); // Read-only file system.
100047     return (NULL);
100048 }
100049 //
100050 // Verify access permissions for the directory. The
100051 // number "3" means
100052 // that the user must have access permission and
100053 // write permission:
100054 // "-wx" == 2+1 == 3.
100055 //
100056 status = inode_check (inode_directory, S_IFDIR, 3,
100057     ps->euid, ps->egid);
100058 if (status != 0)
100059 {
100060     inode_put (inode_directory);
100061     return (NULL);
100062 }
100063 //
100064 // If the inode to link was not specified, it must
100065 // be created.
100066 // From now on, the inode is referenced with the
100067 // variable
100068 // 'inode_new'.
100069 //
100070 inode_new = inode;
100071 //
100072 if (inode_new == NULL)
100073 {
100074     inode_new =
100075         inode_alloc (inode_directory->sb->device, mode,
100076             ps->euid, ps->egid);
100077     if (inode_new == NULL)
100078     {
100079         //
100080         // The inode allocation failed, so, also the
100081         // directory
100082         // must be released, before return.
100083         //
100084         inode_put (inode_directory);
100085         return (NULL);
100086     }
100087 }
100088 //
100089 // Read the directory content and try to add the new
100090 // item.
100091 //
100092 for (start = 0;
100093     start < inode_directory->size;
100094     start += inode_directory->sb->blksize)
100095 {
100096     size_read =
100097         inode_file_read (inode_directory, start,
100098             buffer,
100099             inode_directory->sb->blksize,
100100             NULL);
100101     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:

```

```

108039 // mount point
108040 return (NULL); // already used.
108041 }
108042 //
108043 // If it is a partition, find if it can be mounted.
108044 //
108045 if ((dev_major == DEV_DM_MAJOR) && (p));
108046 {
108047 //
108048 // It is a partition.
108049 //
108050 if (dm_table[d].part[p].type != PART_TYPE_MINIX)
108051 {
108052 {
108053 errset (E_PART_TYPE_NOT_MINIX); // Not Minix!
108054 return (NULL); // Cannot mount.
108055 }
108056 //
108057 // The inode is not yet mounting anything, or it is
108058 // new: find a free
108059 // slot inside the super block table.
108060 //
108061 sb = sb_reference ((dev_t) - 1);
108062 if (sb == NULL)
108063 {
108064 errset (EBUSY); // Device or resource busy:
108065 return (NULL); // no free slots.
108066 }
108067 //
108068 // A free slot was found: the super block header
108069 // must be loaded, but
108070 // before it is necessary to calculate the header
108071 // size to be read.
108072 //
108073 size_sb = offsetof (sb_t, device);
108074 //
108075 // Then fix the starting point.
108076 //
108077 start = 1024; // After boot block.
108078 //
108079 // Read the file system super block header.
108080 //
108081 size_read =
108082 dev_io ((pid_t) 0, device, DEV_READ, start, sb,
108083 size_sb, NULL);
108084 if (size_read != size_sb)
108085 {
108086 errset (EIO); // I/O error.
108087 return (NULL);
108088 }
108089 //
108090 // Save some more data.
108091 //
108092 sb->device = device;
108093 sb->options = options;
108094 sb->inode_mounted_on = *inode_mnt;
108095 sb->blksize = (1024 << sb->log2_size_zone);
108096 //
108097 // Check if the super block data is valid.
108098 //
108099 if (sb->magic_number != 0x137F)
108100 {
108101 errset (ENODEV); // No such device: unsupported
108102 sb->device = 0; // file system type.
108103 return (NULL);
108104 }
108105 if (sb->map_inode_blocks > SB_MAX_INODE_BLOCKS)
108106 {
108107 errset (E_MAP_INODE_TOO_BIG);
108108 return (NULL);
108109 }
108110 if (sb->map_zone_blocks > SB_MAX_ZONE_BLOCKS)
108111 {
108112 errset (E_MAP_ZONE_TOO_BIG);
108113 return (NULL);
108114 }
108115 if (sb->blksize > SB_MAX_ZONE_SIZE)
108116 {
108117 errset (E_DATA_ZONE_TOO_BIG);
108118 return (NULL);
108119 }
108120 //
108121 // A right super block header was loaded from disk,
108122 // now load the super block inode bit map.
108123 //
108124 start = 1024; // After boot block.
108125 start += 1024; // After super block.

```

```

108026 //
108027 // Reset map in memory before loading.
108028 //
108029 for (m = 0; m < SB_MAP_INODE_SIZE; m++) // [2]
108030 {
108031 sb->map_inode[m] = 0xFFFF; // [2]
108032 }
108033 size_map = sb->map_inode_blocks * 1024;
108034 size_read =
108035 dev_io ((pid_t) - 1, sb->device, DEV_READ, start,
108036 sb->map_inode, size_map, NULL);
108037 if (size_read != size_map)
108038 {
108039 errset (EIO); // I/O error.
108040 return (NULL);
108041 }
108042 //
108043 // Load the super block zone bit map.
108044 //
108045 // After boot block:
108046 //
108047 start = 1024;
108048 //
108049 // After the super block:
108050 //
108051 start += 1024;
108052 //
108053 // After inode bit map:
108054 //
108055 start += (sb->map_inode_blocks * 1024);
108056 //
108057 // Reset map in memory, before loading.
108058 //
108059 for (m = 0; m < SB_MAP_ZONE_SIZE; m++)
108060 {
108061 sb->map_zone[m] = 0xFFFF;
108062 }
108063 //
108064 size_map = sb->map_zone_blocks * 1024;
108065 size_read =
108066 dev_io ((pid_t) - 1, sb->device, DEV_READ, start,
108067 sb->map_zone, size_map, NULL);
108068 if (size_read != size_map)
108069 {
108070 errset (EIO); // I/O error.
108071 return (NULL);
108072 }
108073 //
108074 // Check the inode that should mount the super
108075 // block. If '*inode_mnt' is 'NULL', then it is meant
108076 // to be the first mount of the root file system.
108077 // In such case, the inode must be loaded too,
108078 // and the value for '*inode_mnt' must be modified.
108079 //
108080 if (*inode_mnt == NULL)
108081 {
108082 *inode_mnt = inode_get (device, 1);
108083 }
108084 //
108085 // Check for a valid value.
108086 //
108087 if (*inode_mnt == NULL)
108088 {
108089 //
108090 // This is bad!
108091 //
108092 errset (EUNKNOWN); // Unknown error.
108093 return (NULL);
108094 }
108095 //
108096 // A valid inode is available for the mount.
108097 //
108098 (*inode_mnt)->sb_attached = sb;
108099 //
108100 // Update the super block too.
108101 //
108102 sb->inode_mounted_on = *inode_mnt;
108103 //
108104 // Return the super block pointer.
108105 //
108106 return (sb);
108107 }

```



## 94.5.34 kernel/fs/sb\_print.c

« Si veda la sezione 93.6.28.

```

1090001 #include <sys/os32.h>
1090002 #include <kernel/fs.h>
1090003 #include <kernel/lib_k.h>
1090004 //-----
1090005 void
1090006 sb_print (void)
1090007 {
1090008     int s;
1090009     //
1090010     k_printf
1090011     ("      mnt          1st zone max file "
1090012      " inode zone \n");
1090013     k_printf
1090014     ("dev dev inodes blocks dz size size KiB "
1090015      "blocks blocks\n");
1090016     //
1090017     for (s = 0; s < SB_MAX_SLOTS; s++)
1090018     {
1090019         if (sb_table[s].device == 0)
1090020         {
1090021             continue;
1090022         }
1090023         k_printf
1090024         ("%04x %04x % 6i % 6i % 3i "
1090025          "% 4i % 8i % 6i % 6i\n",
1090026          sb_table[s].device,
1090027          sb_table[s].inode_mounted_on->sb_attached->device,
1090028          sb_table[s].inodes, sb_table[s].zones,
1090029          sb_table[s].first_data_zone,
1090030          (1024 << sb_table[s].log2_size_zone),
1090031          sb_table[s].max_file_size / 1024,
1090032          sb_table[s].map_inode_blocks,
1090033          sb_table[s].map_zone_blocks);
1090034     }
1090035 }

```

## 94.5.35 kernel/fs/sb\_reference.c

« Si veda la sezione 93.6.29.

```

1100001 #include <kernel/fs.h>
1100002 #include <errno.h>
1100003 //-----
1100004 sb_t *
1100005 sb_reference (dev_t device)
1100006 {
1100007     int s;          // Slot index.
1100008     //
1100009     // If device is zero, a reference to the whole table
1100010     // is returned.
1100011     //
1100012     if (device == 0)
1100013     {
1100014         return (sb_table);
1100015     }
1100016     //
1100017     // If device is ((dev_t) -1), a reference to a free
1100018     // slot is
1100019     // returned.
1100020     //
1100021     if (device == ((dev_t) - 1))
1100022     {
1100023         for (s = 0; s < SB_MAX_SLOTS; s++)
1100024         {
1100025             if (sb_table[s].device == 0)
1100026             {
1100027                 return (&sb_table[s]);
1100028             }
1100029         }
1100030         return (NULL);
1100031     }
1100032     //
1100033     // A device was selected: find the super block
1100034     // associated to it.
1100035     //
1100036     for (s = 0; s < SB_MAX_SLOTS; s++)
1100037     {
1100038         if (sb_table[s].device == device)
1100039         {
1100040             return (&sb_table[s]);
1100041         }
1100042     }
1100043     //
1100044     // The super block was not found.

```

```

1100045 //
1100046     return (NULL);
1100047 }

```

## 94.5.36 kernel/fs/sb\_save.c

« Si veda la sezione 93.6.30.

```

1110001 #include <kernel/fs.h>
1110002 #include <errno.h>
1110003 #include <kernel/dev.h>
1110004 //-----
1110005 int
1110006 sb_save (sb_t * sb)
1110007 {
1110008     ssize_t size_written;
1110009     addr_t start;
1110010     size_t size_map;
1110011     //
1110012     // Check for valid argument.
1110013     //
1110014     if (sb == NULL)
1110015     {
1110016         errset (EINVAL); // Invalid argument.
1110017         return (-1);
1110018     }
1110019     //
1110020     // Check if the super block changed for some reason
1110021     // (only the inode and the zone maps can change
1110022     // really).
1110023     //
1110024     if (!sb->changed)
1110025     {
1110026         //
1110027         // Nothing to save.
1110028         //
1110029         return (0);
1110030     }
1110031     //
1110032     // Something inside the super block changed: start
1110033     // the procedure to save the inode map (recall that
1110034     // the super block header is not saved, because it
1110035     // never changes).
1110036     //
1110037     start = 1024; // After boot block.
1110038     start += 1024; // After super block.
1110039     size_map = sb->map_inode_blocks * 1024;
1110040     size_written =
1110041     dev_io ((pid_t) - 1, sb->device, DEV_WRITE, start,
1110042            sb->map_inode, size_map, NULL);
1110043     if (size_written != size_map)
1110044     {
1110045         //
1110046         // Error writing the map.
1110047         //
1110048         errset (EIO); // I/O error.
1110049         return (-1);
1110050     }
1110051     //
1110052     // Start the procedure to save the zone map.
1110053     //
1110054     start = 1024; // After boot block.
1110055     start += 1024; // After super block.
1110056     start += (sb->map_inode_blocks * 1024); // After
1110057     // inode bit
1110058     // map.
1110059     size_map = sb->map_zone_blocks * 1024;
1110060     size_written =
1110061     dev_io ((pid_t) - 1, sb->device, DEV_WRITE, start,
1110062            sb->map_zone, size_map, NULL);
1110063     if (size_written != size_map)
1110064     {
1110065         //
1110066         // Error writing the map.
1110067         //
1110068         errset (EIO); // I/O error.
1110069         return (-1);
1110070     }
1110071     //
1110072     // Super block saved.
1110073     //
1110074     sb->changed = 0;
1110075     //
1110076     return (0);
1110077 }

```

## 94.5.37 kernel/fs/sb\_zone\_status.c

Si veda la sezione 93.6.26.

```

1120001 #include <kernel/fs.h>
1120002 #include <errno.h>
1120003 -----
1120004 int
1120005 sb_zone_status (sb_t * sb, zno_t zone)
1120006 {
1120007     int map_element;
1120008     int map_bit;
1120009     int map_mask;
1120010     //
1120011     // Check arguments.
1120012     //
1120013     if (zone == 0 || sb == NULL)
1120014     {
1120015         errset (EINVAL); // Invalid argument.
1120016         return (-1);
1120017     }
1120018     //
1120019     // Calculate the map element, the map bit and the
1120020     // map mask.
1120021     //
1120022     map_element = zone / 16;
1120023     map_bit = zone % 16;
1120024     map_mask = 1 << map_bit;
1120025     //
1120026     // Check the zone and return.
1120027     //
1120028     if (sb->map_zone[map_element] & map_mask)
1120029     {
1120030         return (1); // True.
1120031     }
1120032     else
1120033     {
1120034         return (0); // False.
1120035     }
1120036 }

```

## 94.5.38 kernel/fs/sock\_free\_port.c

Si veda la sezione 93.6.32.

```

1130001 #include <kernel/fs.h>
1130002 #include <errno.h>
1130003 #include <fcntl.h>
1130004 -----
1130005 h_port_t
1130006 sock_free_port (void)
1130007 {
1130008     int skn;
1130009     h_port_t lport;
1130010     //
1130011     for (lport = 0xFFFF; lport >= 1024; lport--)
1130012     {
1130013         for (skn = 0; skn < SOCK_MAX_SLOTS; skn++)
1130014         {
1130015             if (sock_table[skn].lport == lport)
1130016             {
1130017                 //
1130018                 // The port is used.
1130019                 //
1130020                 break;
1130021             }
1130022         }
1130023         if (sock_table[skn].lport != lport)
1130024         {
1130025             //
1130026             // The port is new.
1130027             //
1130028             return (lport);
1130029         }
1130030     }
1130031     //
1130032     // If we are here, no free port was found.
1130033     //
1130034     return ((h_port_t) 0);
1130035 }

```

## 94.5.39 kernel/fs/sock\_reference.c

Si veda la sezione 93.6.33.

```

1140001 #include <kernel/fs.h>
1140002 #include <errno.h>
1140003 #include <fcntl.h>

```

```

1140004 -----
1140005 sock_t *
1140006 sock_reference (int skn)
1140007 {
1140008     //
1140009     // Check type of request.
1140010     //
1140011     if (skn < 0)
1140012     {
1140013         //
1140014         // Find a free slot.
1140015         //
1140016         for (skn = 0; skn < SOCK_MAX_SLOTS; skn++)
1140017         {
1140018             if (sock_table[skn].active == 0)
1140019             {
1140020                 return (&sock_table[skn]);
1140021             }
1140022         }
1140023         return (NULL);
1140024     }
1140025     else if (skn > SOCK_MAX_SLOTS)
1140026     {
1140027         return (NULL);
1140028     }
1140029     else
1140030     {
1140031         return (&sock_table[skn]);
1140032     }
1140033 }

```

## 94.5.40 kernel/fs/zone\_alloc.c

Si veda la sezione 93.6.34.

```

1150001 #include <kernel/fs.h>
1150002 #include <kernel/dev.h>
1150003 #include <errno.h>
1150004 -----
1150005 zno_t
1150006 zone_alloc (sb_t * sb)
1150007 {
1150008     int m; // Index inside the inode map.
1150009     int map_element;
1150010     int map_bit;
1150011     int map_mask;
1150012     zno_t zone;
1150013     char buffer[SB_MAX_ZONE_SIZE];
1150014     int status;
1150015     //
1150016     // Verify if write is allowed.
1150017     //
1150018     if (sb->options & MOUNT_RO)
1150019     {
1150020         errset (EROFS); // Read-only file system.
1150021         return ((zno_t) 0);
1150022     }
1150023     //
1150024     // Write allowed: scan the zone map, to find a free
1150025     // zone. If a free zone can be found, allocate it
1150026     // inside the map.
1150027     // Index 'm' starts from one, because the first bit
1150028     // of the map is reserved for a 'zero' data-zone
1150029     // that does not exist: the second bit is for the
1150030     // real first data-zone.
1150031     //
1150032     for (zone = 0, m = 1; m < (SB_MAP_ZONE_SIZE * 16); m++)
1150033     {
1150034         map_element = m / 16;
1150035         map_bit = m % 16;
1150036         map_mask = 1 << map_bit;
1150037         if (!(sb->map_zone[map_element] & map_mask))
1150038         {
1150039             //
1150040             // Found a free place: set the map.
1150041             //
1150042             sb->map_zone[map_element] |= map_mask;
1150043             sb->changed = 1;
1150044             //
1150045             // The *second* bit inside the map is for
1150046             // the first data zone (the zone after the
1150047             // inode table inside the file system),
1150048             // because the first is for a special
1150049             // 'zero' data zone, not really used.
1150050             //
1150051             zone = sb->first_data_zone + m - 1; // Found
1150052             // a free

```

```

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

```

## 94.5.41 kernel/fs/zone\_free.c

«

Si veda la sezione 93.6.34.

```

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

```

```

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

```

## 94.5.42 kernel/fs/zone\_print.c

Si veda la sezione 93.6.36.

«

```

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

```

## 94.5.43 kernel/fs/zone\_read.c

Si veda la sezione 93.6.37.

«

```

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

```

```

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

```

#### 94.5.44 kernel/fs/zone\_write.c

Si veda la sezione 93.6.37.

```

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

```

#### 94.6 os32: «kernel/ibm\_i386.h»

Si veda la sezione 93.7.

```

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

```

```

1200006 #include <stdbool.h>
1200007 #include <stdarg.h>
1200008 //-----
1200009 // GDT
1200010 //-----
1200011 #define GDT_ITEMS      256 // Max is 8192 items.
1200012 //
1200013 typedef struct
1200014 {
1200015     uint32_t limit_a:16, base_a:16;
1200016     uint32_t base_b:8,
1200017     accessed:1,
1200018     write_execute:1,
1200019     expansion_conforming:1,
1200020     code_or_data:1,
1200021     code_data_or_system:1,
1200022     dpl:2,
1200023     present:1,
1200024     limit_b:4,
1200025     available:1, reserved:1, big:1, granularity:1, base_c:8;
1200026 } gdt_t;
1200027 //
1200028 extern gdt_t gdt_table[GDT_ITEMS];
1200029 //-----
1200030 typedef struct
1200031 {
1200032     uint16_t limit;
1200033     uint32_t base;
1200034 } __attribute__((packed)) gdtr_t; // [1]
1200035 //
1200036 extern gdtr_t gdtr_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 idtr_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,

```



```

1210016      # used.
1210017      mov $0, %eax      # Reset EAX.
1210018      in  %dx, %eax     # 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

&lt;&lt;

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

&lt;&lt;

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

&lt;&lt;

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

&lt;&lt;

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

&lt;&lt;

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

&lt;&lt;

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.

```

128001 #include <kernel/ibm_i386.h>
128002 #include <kernel/multiboot.h>
128003 //-----
128004 void
128005 gdt (void)
128006 {
128007     uint32_t blocks;    // Total available memory
128008     // blocks.
128009     //
128010     // Calculate memory blocks.
128011     //
128012     blocks = (multiboot.mem_upper * 1024) / 4096;
128013     //
128014     // Set data for GDTR register.
128015     //
128016     gdt_register.limit = (sizeof (gdt_table) - 1);
128017     gdt_register.base = (uint32_t) &gdt_table[0];
128018     //
128019     // Reset items inside 'gdt_table[]'.
128020     // gdt_table[0] must be null and is not to be
128021     // used.
128022     //
128023     int i;
128024     for (i = 0; i < GDT_ITEMS; i++)
128025     {
128026         gdt_segment (i, 0, 0, 0, 0);
128027     }
128028     //
128029     // gdt_table[1] is for kernel code.
128030     // It covers all the available memory, with DPL 0.
128031     // The selector is 8+0 = 0x08+0.
128032     //
128033     gdt_segment (1, 0, blocks, 1, 1, 0);
128034     //
128035     // gdt_table[2] is for kernel data, including stack
128036     // (the stack
128037     // MUST be in the same address space of data).
128038     // It covers all the available memory, with DPL 0.
128039     // The selector is 16+0 = 0x10+0.
128040     //
128041     gdt_segment (2, 0, blocks, 1, 0, 0);
128042     //
128043     // Load the GDT table.
128044     //
128045     gdt_load (&gdt_register);
128046 }

```

## 94.6.9 kernel/ibm\_i386/gdt\_load.s

Si veda la sezione 93.7.

```

128001 .globl gdt_load
128002 #
128003 gdt_load:
128004     enter $0, $0
128005     .equ gdtr_pointer, 8           # Primo argomento.
128006     mov gdtr_pointer(%ebp), %eax   # Copia il
128007                                     # puntatore in EAX.
128008     leave
128009     #
128010     lgdt (%eax)                  # Carica il registro GDTR
128011                                     # dall'indirizzo in EAX.
128012     #
128013     # 2 kernel data, included stack, DPL 0, covering
128014     # all the available
128015     # memory: segment selector 0x10+0.
128016     #
128017     mov $16, %ax
128018     mov %ax, %ds
128019     mov %ax, %es
128020     mov %ax, %fs
128021     mov %ax, %gs
128022     mov %ax, %ss                # The stack MUST be in the same
128023                                     # address space of the other
128024                                     # data, to allow pointers to
128025                                     # work correctly.
128026     #
128027     # 2 kernel code, DPL 0, covering all the available
128028     # memory:
128029     # segment selector 0x08+0.
128030     #
128031     jmp $8, $flush
128032 flush:
128033     ret

```

## 94.6.10 kernel/ibm\_i386/gdt\_print.c

Si veda la sezione 93.7.

```

130001 #include <kernel/ibm_i386.h>
130002 #include <kernel/lib_k.h>
130003 //
130004 void
130005 gdt_print (void *gdtr, unsigned int first,
130006           unsigned int last)
130007 {
130008     gdtr_t *g = gdtr;
130009     uint32_t *p = (uint32_t *) g->base;
130010     //
130011     int max = (g->limit + 1) / (sizeof (uint32_t));
130012     int i;
130013     //
130014     if (((first * 2) > max) || (first > last))
130015     {
130016         return;
130017     }
130018     //
130019     k_printf ("%s base: 0x%08" PRIx32 " limit: 0x%04"
130020              " PRiX32 "\n", __func__, g->base, g->limit);
130021     //
130022     for (i = (first * 2); i < max && i <= (last * 2); i += 2)
130023     {
130024         k_printf ("[%4" PRIx32 "] %032" PRIb32 " %032"
130025                  PRIb32 "\n", i / 2, p[i], p[i + 1]);
130026     }
130027 }

```

## 94.6.11 kernel/ibm\_i386/gdt\_public.c

Si veda la sezione 93.7.

```

131001 #include <kernel/ibm_i386.h>
131002 //-----
131003 gdt_t gdt_table[GDT_ITEMS];
131004 gdtr_t gdt_register;

```

## 94.6.12 kernel/ibm\_i386/gdt\_segment.c

Si veda la sezione 93.7.

```

132001 #include <kernel/ibm_i386.h>
132002 #include <errno.h>
132003 //-----
132004 int
132005 gdt_segment (int segment,
132006             uint32_t base,
132007             uint32_t limit,
132008             bool present, bool code, unsigned char dpl)
132009 {
132010     //
132011     // Verify if the segment is valid.
132012     //
132013     if ((segment >= ((sizeof (gdt_table)) / 8))
132014         || (segment < 0))
132015     {
132016         errset (EINVAL);
132017         return (-1);
132018     }
132019     //
132020     // Limit.
132021     //
132022     gdt_table[segment].limit_a = (limit & 0x0000FFFF);
132023     gdt_table[segment].limit_b = limit / 0x10000;
132024     //
132025     // Base address.
132026     //
132027     gdt_table[segment].base_a = (base & 0x0000FFFF);
132028     gdt_table[segment].base_b =
132029         ((base / 0x10000) & 0x000000FF);
132030     gdt_table[segment].base_c = (base / 0x1000000);
132031     //
132032     // Attributes.
132033     //
132034     //
132035     // Internal update.
132036     //
132037     gdt_table[segment].accessed = 0;
132038     //
132039     // r, w, x.
132040     //
132041     gdt_table[segment].write_execute = 1;
132042     //
132043 }

```

```

132044 // Normal and conforming.
132045 //
132046 gdt_table[segment].expansion_conforming = 0;
132047 //
132048 gdt_table[segment].code_or_data = code;
132049 gdt_table[segment].code_data_or_system = 1;
132050 gdt_table[segment].dpl = dpl;
132051 gdt_table[segment].present = present;
132052 gdt_table[segment].available = 0; // 0
132053 gdt_table[segment].reserved = 0; // 0
132054 gdt_table[segment].big = 1; // 32 bit
132055 gdt_table[segment].granularity = 1; // 4 Kibyte
132056 //
132057 return (0);
132058 }

```

#### 94.6.13 kernel/ibm\_i386/idt.c

Si veda la sezione 93.7.

```

133001 #include <kernel/ibm_i386.h>
133002 //-----
133003 void
133004 idt (void)
133005 {
133006 //
133007 // Set necessary data for the IDTR register.
133008 //
133009 idt_register.limit = (sizeof (idt_table) - 1);
133010 idt_register.base = (uint32_t) & idt_table[0];
133011 //
133012 // Reset all items inside the array 'idt_table[]'.
133013 //
133014 int i;
133015 for (i = 0; i < IDT_ITEMS; i++)
133016 {
133017     idt_descriptor (i, 0, 0, 0, 0, 0);
133018 }
133019 //
133020 // Place hardware interrupt from IRQ 0 to IRQ 7
133021 // starting from descriptor 32 and from IRQ 8 to
133022 // IRQ 15 starting from descriptor 40.
133023 //
133024 idt_irq_remap (32, 40);
133025 //
133026 // Set the ISR routines to the items inside the IDT
133027 // table.
133028 //
133029 idt_descriptor (0, isr_0, 0x0008, 1, 0xE, 0);
133030 idt_descriptor (1, isr_1, 0x0008, 1, 0xE, 0);
133031 idt_descriptor (2, isr_2, 0x0008, 1, 0xE, 0);
133032 idt_descriptor (3, isr_3, 0x0008, 1, 0xE, 0);
133033 idt_descriptor (4, isr_4, 0x0008, 1, 0xE, 0);
133034 idt_descriptor (5, isr_5, 0x0008, 1, 0xE, 0);
133035 idt_descriptor (6, isr_6, 0x0008, 1, 0xE, 0);
133036 idt_descriptor (7, isr_7, 0x0008, 1, 0xE, 0);
133037 idt_descriptor (8, isr_8, 0x0008, 1, 0xE, 0);
133038 idt_descriptor (9, isr_9, 0x0008, 1, 0xE, 0);
133039 idt_descriptor (10, isr_10, 0x0008, 1, 0xE, 0);
133040 idt_descriptor (11, isr_11, 0x0008, 1, 0xE, 0);
133041 idt_descriptor (12, isr_12, 0x0008, 1, 0xE, 0);
133042 idt_descriptor (13, isr_13, 0x0008, 1, 0xE, 0);
133043 idt_descriptor (14, isr_14, 0x0008, 1, 0xE, 0);
133044 idt_descriptor (15, isr_15, 0x0008, 1, 0xE, 0);
133045 idt_descriptor (16, isr_16, 0x0008, 1, 0xE, 0);
133046 idt_descriptor (17, isr_17, 0x0008, 1, 0xE, 0);
133047 idt_descriptor (18, isr_18, 0x0008, 1, 0xE, 0);
133048 idt_descriptor (19, isr_19, 0x0008, 1, 0xE, 0);
133049 idt_descriptor (20, isr_20, 0x0008, 1, 0xE, 0);
133050 idt_descriptor (21, isr_21, 0x0008, 1, 0xE, 0);
133051 idt_descriptor (22, isr_22, 0x0008, 1, 0xE, 0);
133052 idt_descriptor (23, isr_23, 0x0008, 1, 0xE, 0);
133053 idt_descriptor (24, isr_24, 0x0008, 1, 0xE, 0);
133054 idt_descriptor (25, isr_25, 0x0008, 1, 0xE, 0);
133055 idt_descriptor (26, isr_26, 0x0008, 1, 0xE, 0);
133056 idt_descriptor (27, isr_27, 0x0008, 1, 0xE, 0);
133057 idt_descriptor (28, isr_28, 0x0008, 1, 0xE, 0);
133058 idt_descriptor (29, isr_29, 0x0008, 1, 0xE, 0);
133059 idt_descriptor (30, isr_10, 0x0008, 1, 0xE, 0);
133060 idt_descriptor (31, isr_31, 0x0008, 1, 0xE, 0);
133061 idt_descriptor (32, isr_32, 0x0008, 1, 0xE, 0);
133062 idt_descriptor (33, isr_33, 0x0008, 1, 0xE, 0);
133063 idt_descriptor (34, isr_34, 0x0008, 1, 0xE, 0);
133064 idt_descriptor (35, isr_35, 0x0008, 1, 0xE, 0);
133065 idt_descriptor (36, isr_36, 0x0008, 1, 0xE, 0);
133066 idt_descriptor (37, isr_37, 0x0008, 1, 0xE, 0);
133067 idt_descriptor (38, isr_38, 0x0008, 1, 0xE, 0);

```

```

133068 idt_descriptor (39, isr_39, 0x0008, 1, 0xE, 0);
133069 idt_descriptor (40, isr_40, 0x0008, 1, 0xE, 0);
133070 idt_descriptor (41, isr_41, 0x0008, 1, 0xE, 0);
133071 idt_descriptor (42, isr_42, 0x0008, 1, 0xE, 0);
133072 idt_descriptor (43, isr_43, 0x0008, 1, 0xE, 0);
133073 idt_descriptor (44, isr_44, 0x0008, 1, 0xE, 0);
133074 idt_descriptor (45, isr_45, 0x0008, 1, 0xE, 0);
133075 idt_descriptor (46, isr_46, 0x0008, 1, 0xE, 0);
133076 idt_descriptor (47, isr_47, 0x0008, 1, 0xE, 0);
133077 //
133078 // The following item is for the system calls.
133079 //
133080 idt_descriptor (128, isr_128, 0x0008, 1, 0xE, 0);
133081 //
133082 // Activate the IDT loading the IDTR register.
133083 //
133084 idt_load (&idt_register);
133085 }

```

#### 94.6.14 kernel/ibm\_i386/idt\_descriptor.c

Si veda la sezione 93.7.

```

134001 #include <kernel/ibm_i386.h>
134002 //-----
134003 void
134004 idt_descriptor (int desc,
134005                void *isr,
134006                uint16_t selector,
134007                bool present, char type, char dpl)
134008 {
134009     uint32_t offset = (uint32_t) isr;
134010 //
134011 // Unset reserved bits and the system bit.
134012 //
134013 idt_table[desc].filler = 0;
134014 idt_table[desc].system = 0;
134015 //
134016 // Relative address.
134017 //
134018 idt_table[desc].offset_a = (offset & 0x0000FFFF);
134019 idt_table[desc].offset_b = (offset / 0x10000);
134020 //
134021 // Selector.
134022 //
134023 idt_table[desc].selector = selector;
134024 //
134025 // Valid item?
134026 //
134027 idt_table[desc].present = present;
134028 //
134029 // Type (gate type).
134030 //
134031 idt_table[desc].type = (type & 0x0F);
134032 //
134033 // DPL.
134034 //
134035 idt_table[desc].dpl = (dpl & 0x03);
134036 }

```

#### 94.6.15 kernel/ibm\_i386/idt\_irq\_remap.c

Si veda la sezione 93.7.

```

135001 #include <kernel/lib_k.h>
135002 #include <kernel/ibm_i386.h>
135003 //-----
135004 #define DEBUG 0
135005 //-----
135006 void
135007 idt_irq_remap (unsigned int offset_1, unsigned int offset_2)
135008 {
135009 //
135010 // PIC_P è il PIC primario o «master»;
135011 // PIC_S è il PIC secondario o «slave».
135012 //
135013 // Quando si manifesta un IRQ che riguarda il PIC
135014 // secondario,
135015 // il PIC primario riceve IRQ 2.
135016 //
135017 // ICW = initialization command word.
135018 // OCW = operation command word.
135019 //
135020 if (DEBUG)
135021 {
135022     k_printf
135023         ("%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 idtr_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.

```

1440001 .extern isr_exception_unrecoverable
1440002 .extern isr_irq_clear
1440003 .extern isr_irq_clear_pic1
1440004 .extern isr_irq_clear_pic2
1440005 .extern kbd_isr
1440006 .extern sysroutine
1440007 .extern proc_current
1440008 .extern proc_stack_segment_selector;
1440009 .extern proc_stack_pointer
1440010 .extern proc_scheduler
1440011 #.extern proc_sch_terminals
1440012 #
1440013 .global _clock_kernel
1440014 .global _clock_time
1440015 .global _ksp
1440016 #
1440017 .global isr_0
1440018 .global isr_1
1440019 .global isr_2
1440020 .global isr_3
1440021 .global isr_4
1440022 .global isr_5
1440023 .global isr_6
1440024 .global isr_7
1440025 .global isr_8
1440026 .global isr_9
1440027 .global isr_10
1440028 .global isr_11
1440029 .global isr_12
1440030 .global isr_13
1440031 .global isr_14
1440032 .global isr_15
1440033 .global isr_16
1440034 .global isr_17
1440035 .global isr_18
1440036 .global isr_19
1440037 .global isr_20
1440038 .global isr_21
1440039 .global isr_22
1440040 .global isr_23
1440041 .global isr_24
1440042 .global isr_25
1440043 .global isr_26
1440044 .global isr_27
1440045 .global isr_28
1440046 .global isr_29
1440047 .global isr_30
1440048 .global isr_31
1440049 .global isr_32
1440050 .global isr_33
1440051 .global isr_34
1440052 .global isr_35
1440053 .global isr_36
1440054 .global isr_37
1440055 .global isr_38
1440056 .global isr_39
1440057 .global isr_40
1440058 .global isr_41
1440059 .global isr_42
1440060 .global isr_43
1440061 .global isr_44
1440062 .global isr_45
1440063 .global isr_46
1440064 .global isr_47
1440065 .global isr_128
1440066 #-----
1440067 .section .data
1440068 #-----
1440069 proc_syscallnr:      .int 0x00000000
1440070 proc_msg_offset:    .int 0x00000000
1440071 proc_msg_size:      .int 0x00000000
1440072 proc_instruction_pointer: .int 0x00000000
1440073 proc_back_address:  .int 0x00000000
1440074 _ksp:                .int 0x00000000
1440075 syscall_working:    .int 0x00000000
1440076 _clock_kernel:      .int 0x00000000
1440077 kticks_lo:          .int 0x00000000
1440078 kticks_hi:          .int 0x00000000
1440079 _clock_time:        .int 0x00000000
1440080 tticks_lo:          .int 0x00000000
1440081 tticks_hi:          .int 0x00000000
1440082 #-----
1440083 .section .text
1440084 #-----
1440085 #

```

```

1440086 # Inside the stack there is already, placed by the CPU:
1440087 # [omissis]
1440088 # push %eflags
1440089 # push %cs
1440090 # push %eip
1440091 #
1440092 #-----
1440093 isr_0:      # «division by zero exception»
1440094 cli
1440095 push $0    # Null error code.
1440096 push $0    # Exception number.
1440097 jmp exception_unrecoverable
1440098 #-----
1440099 isr_1:      # «debug exception»
1440100 cli
1440101 push $0    # Null error code.
1440102 push $1    # Exception number.
1440103 jmp exception_unrecoverable
1440104 #-----
1440105 isr_2:      # «non maskable interrupt exception»
1440106 cli
1440107 push $0    # Null error code.
1440108 push $2    # Exception number.
1440109 jmp exception_unrecoverable
1440110 #-----
1440111 isr_3:      # «breakpoint exception»
1440112 cli
1440113 push $0    # Null error code.
1440114 push $3    # Exception number.
1440115 jmp exception_unrecoverable
1440116 #-----
1440117 isr_4:      # «into detected overflow exception»
1440118 cli
1440119 push $0    # Null error code.
1440120 push $4    # Exception number.
1440121 jmp exception_unrecoverable
1440122 #-----
1440123 isr_5:      # «out of bounds exception»
1440124 cli
1440125 push $0    # Null error code.
1440126 push $5    # Exception number.
1440127 jmp exception_unrecoverable
1440128 #-----
1440129 isr_6:      # «invalid opcode exception»
1440130 cli
1440131 push $0    # Null error code.
1440132 push $6    # Exception number.
1440133 jmp exception_unrecoverable
1440134 #-----
1440135 isr_7:      # «no coprocessor exception»
1440136 cli
1440137 push $0    # Null error code.
1440138 push $7    # Exception number.
1440139 jmp exception_unrecoverable
1440140 #-----
1440141 isr_8:      # «double fault exception»
1440142 cli
1440143 #          # Error code already present.
1440144 push $8    # Exception number.
1440145 jmp exception_unrecoverable
1440146 #-----
1440147 isr_9:      # «coprocessor segment overrun
1440148 #          # exception»
1440149 cli
1440150 push $0    # Null error code.
1440151 push $9    # Exception number.
1440152 jmp exception_unrecoverable
1440153 #-----
1440154 isr_10:     # «bad TSS exception»
1440155 cli
1440156 #          # Error code already present.
1440157 push $10   # Exception number.
1440158 jmp exception_unrecoverable
1440159 #-----
1440160 isr_11:     # «segment not present exception»
1440161 cli
1440162 #          # Error code already present.
1440163 push $11   # Exception number.
1440164 jmp exception_unrecoverable
1440165 #-----
1440166 isr_12:     # «stack fault exception»
1440167 cli
1440168 #          # Error code already present.
1440169 push $12   # Exception number.
1440170 jmp exception_unrecoverable
1440171 #-----
1440172 isr_13:     # «general protection fault exception»

```

```

1440173 cli
1440174 # # Error code already present.
1440175 push $13 # Exception number.
1440176 jmp exception_unrecoverable
1440177 #-----
1440178 isr_14: # <page fault exception>
1440179 cli
1440180 # # Error code already present.
1440181 push $14 # Exception number.
1440182 jmp exception_unrecoverable
1440183 #-----
1440184 isr_15: # <unknown interrupt exception>
1440185 cli
1440186 push $0 # Null error code.
1440187 push $15 # Exception number.
1440188 jmp exception_unrecoverable
1440189 #-----
1440190 isr_16: # <coprocessor fault exception>
1440191 cli
1440192 push $0 # Null error code.
1440193 push $16 # Exception number.
1440194 jmp exception_unrecoverable
1440195 #-----
1440196 isr_17: # <alignment check exception>
1440197 cli
1440198 push $0 # Null error code.
1440199 push $17 # Exception number.
1440200 jmp exception_unrecoverable
1440201 #-----
1440202 isr_18: # <machine check exception>
1440203 cli
1440204 push $0 # Null error code.
1440205 push $18 # Exception number.
1440206 jmp exception_unrecoverable
1440207 #-----
1440208 isr_19: # <reserved exception>
1440209 cli
1440210 push $0 # Null error code.
1440211 push $19 # Exception number.
1440212 jmp exception_unrecoverable
1440213 #-----
1440214 isr_20: # <reserved exception>
1440215 cli
1440216 push $0 # Null error code.
1440217 push $20 # Exception number.
1440218 jmp exception_unrecoverable
1440219 #-----
1440220 isr_21: # <reserved exception>
1440221 cli
1440222 push $0 # Null error code.
1440223 push $21 # Exception number.
1440224 jmp exception_unrecoverable
1440225 #-----
1440226 isr_22: # <reserved exception>
1440227 cli
1440228 push $0 # Null error code.
1440229 push $22 # Exception number.
1440230 jmp exception_unrecoverable
1440231 #-----
1440232 isr_23: # <reserved exception>
1440233 cli
1440234 push $0 # Null error code.
1440235 push $23 # Exception number.
1440236 jmp exception_unrecoverable
1440237 #-----
1440238 isr_24: # <reserved exception>
1440239 cli
1440240 push $0 # Null error code.
1440241 push $24 # Exception number.
1440242 jmp exception_unrecoverable
1440243 #-----
1440244 isr_25: # <reserved exception>
1440245 cli
1440246 push $0 # Null error code.
1440247 push $25 # Exception number.
1440248 jmp exception_unrecoverable
1440249 #-----
1440250 isr_26: # <reserved exception>
1440251 cli
1440252 push $0 # Null error code.
1440253 push $26 # Exception number.
1440254 jmp exception_unrecoverable
1440255 #-----
1440256 isr_27: # <reserved exception>
1440257 cli
1440258 push $0 # Null error code.
1440259 push $27 # Exception number.

```

```

1440260 jmp exception_unrecoverable
1440261 #-----
1440262 isr_28: # <reserved exception>
1440263 cli
1440264 push $0 # Null error code.
1440265 push $28 # Exception number.
1440266 jmp exception_unrecoverable
1440267 #-----
1440268 isr_29: # <reserved exception>
1440269 cli
1440270 push $0 # Null error code.
1440271 push $29 # Exception number.
1440272 jmp exception_unrecoverable
1440273 #-----
1440274 isr_30: # <reserved exception>
1440275 cli
1440276 push $0 # Null error code.
1440277 push $30 # Exception number.
1440278 jmp exception_unrecoverable
1440279 #-----
1440280 isr_31: # <reserved exception>
1440281 cli
1440282 push $0 # Null error code.
1440283 push $31 # Exception number.
1440284 jmp exception_unrecoverable
1440285 #-----
1440286 isr_32: # IRQ 0: <timer>
1440287 cli
1440288 jmp irq_timer
1440289 #-----
1440290 isr_33: # IRQ 1: tastiera
1440291 cli
1440292 jmp irq_keyboard
1440293 #-----
1440294 isr_34: # IRQ 2: it is fired for IRQ 8 to 15.
1440295 cli
1440296 #
1440297 # IRQ 2 must be ON inside the file
1440298 # 'kernel/proc/proc_init.c', so
1440299 # that it is guaranteed that the PIC 1 is reset
1440300 # here.
1440301 #
1440302 call isr_irq_clear_pic1
1440303 #
1440304 # For IRQ 2 there is nothing else to do, because it
1440305 # is a link to the PIC 2 (IRQ 8 to 15).
1440306 #
1440307 iret
1440308 #-----
1440309 isr_35: # IRQ 3
1440310 cli
1440311 jmp irq_pic1
1440312 #-----
1440313 isr_36: # IRQ 4
1440314 cli
1440315 jmp irq_pic1
1440316 #-----
1440317 isr_37: # IRQ 5
1440318 cli
1440319 jmp irq_pic1
1440320 #-----
1440321 isr_38: # IRQ 6: floppy disk drive
1440322 cli
1440323 jmp irq_pic1
1440324 #-----
1440325 isr_39: # IRQ 7: LPT 1
1440326 cli
1440327 jmp irq_pic1
1440328 #-----
1440329 isr_40: # IRQ 8: <real time clock (RTC)>
1440330 cli
1440331 jmp irq_pic2
1440332 #-----
1440333 isr_41: # IRQ 9
1440334 cli
1440335 jmp irq_pic2
1440336 #-----
1440337 isr_42: # IRQ 10
1440338 cli
1440339 jmp irq_pic2
1440340 #-----
1440341 isr_43: # IRQ 11
1440342 cli
1440343 jmp irq_pic2
1440344 #-----
1440345 isr_44: # IRQ 12: mouse PS/2
1440346 cli

```

```

1410347      jmp irq_pic2
1410348      #-----
1410349      isr_45:      # IRQ 13: math coprocessor
1410350          cli
1410351          jmp irq_pic2
1410352      #-----
1410353      isr_46:      # IRQ 14: primary IDE channel
1410354          cli
1410355          jmp irq_pic2
1410356      #-----
1410357      isr_47:      # IRQ 15: secondary IDE channel
1410358          cli
1410359          jmp irq_pic2
1410360      #-----
1410361      #
1410362      # Unrecoverable exceptions.
1410363      #
1410364      exception_unrecoverable:
1410365          #
1410366          # Previous pushes:
1410367          # [omissis]
1410368          # push %eflags
1410369          # push %cs
1410370          # push %eip
1410371          #
1410372          # push $<error_number>
1410373          # push $<idt_item_number>
1410374          #
1410375          pushl %gs
1410376          pushl %fs
1410377          pushl %es
1410378          pushl %ds
1410379          pushl %edi
1410380          pushl %esi
1410381          pushl %ebp
1410382          pushl %ebx
1410383          pushl %edx
1410384          pushl %ecx
1410385          pushl %eax
1410386          #
1410387          call isr_exception_unrecoverable
1410388          #
1410389          popl %eax
1410390          popl %ecx
1410391          popl %edx
1410392          popl %ebx
1410393          popl %ebp
1410394          popl %esi
1410395          popl %edi
1410396          popl %ds
1410397          popl %es
1410398          popl %fs
1410399          popl %gs
1410400          #
1410401          # Remove the IDT item number and the error code.
1410402          #
1410403          add $4, %esp
1410404          add $4, %esp
1410405          #
1410406          # Return from interrupt.
1410407          #
1410408          iret
1410409      #-----
1410410      #
1410411      # Unspecified IRQ. Currently unused.
1410412      #
1410413      irq:
1410414          #
1410415          # Previous pushes:
1410416          # [omissis]
1410417          # push %eflags
1410418          # push %cs
1410419          # push %eip
1410420          #
1410421          # push $0
1410422          # push $<idt_item_number>
1410423          #
1410424          call isr_irq_clear
1410425          #
1410426          # Remove the IDT item number and the null error
1410427          # code.
1410428          #
1410429          add $4, %esp
1410430          add $4, %esp
1410431          #
1410432          # Return from interrupt.
1410433          #

```

```

1410434          iret
1410435      #-----
1410436      #
1410437      # Keyboard IRQ.
1410438      #
1410439      irq_keyboard:
1410440          #
1410441          # Previous pushes:
1410442          # [omissis]
1410443          # push %eflags
1410444          # push %cs
1410445          # push %eip
1410446          #
1410447          pushl %gs
1410448          pushl %fs
1410449          pushl %es
1410450          pushl %ds
1410451          pushl %edi
1410452          pushl %esi
1410453          pushl %ebp
1410454          pushl %ebx
1410455          pushl %edx
1410456          pushl %ecx
1410457          pushl %eax
1410458          #
1410459          # Set the data segments to the kernel data segment,
1410460          # so that the following variables can be accessed.
1410461          #
1410462          mov $16, %ax # DS, ES, FS and GS.
1410463          mov %ax, %ds
1410464          mov %ax, %es
1410465          mov %ax, %fs
1410466          mov %ax, %gs
1410467          #
1410468          # Check if a system call is already working: if so,
1410469          # just leave (go to L1).
1410470          #
1410471          cmpl $1, syscall_working
1410472          je L1
1410473          #
1410474          # Call the keyboard handler.
1410475          #
1410476          call kbd_isr
1410477          #
1410478          L1: # Restore original registers and return.
1410479          #
1410480          jmp irq_pic1_pop_iret
1410481      #-----
1410482      #
1410483      # Generic IRQ from PIC 1
1410484      #
1410485      irq_pic1:
1410486          #
1410487          # Previous pushes:
1410488          # [omissis]
1410489          # push %eflags
1410490          # push %cs
1410491          # push %eip
1410492          #
1410493          pushl %gs
1410494          pushl %fs
1410495          pushl %es
1410496          pushl %ds
1410497          pushl %edi
1410498          pushl %esi
1410499          pushl %ebp
1410500          pushl %ebx
1410501          pushl %edx
1410502          pushl %ecx
1410503          pushl %eax
1410504          #
1410505          # Set the data segments to the kernel data segment,
1410506          # so that the following variables can be accessed.
1410507          #
1410508          mov $16, %ax # DS, ES, FS and GS.
1410509          mov %ax, %ds
1410510          mov %ax, %es
1410511          mov %ax, %fs
1410512          mov %ax, %gs
1410513          #
1410514          # Check if a system call is already working: if so,
1410515          # just leave (go to L2).
1410516          #
1410517          cmpl $1, syscall_working
1410518          je L2
1410519          #
1410520          # If we are here, no system call is working and a

```

```

1410521 # user process was interrupted.
1410522 # Save process stack registers into kernel data
1410523 # segment.
1410524 #
1410525 mov %ss, proc_stack_segment_selector
1410526 mov %esp, proc_stack_pointer
1410527 #
1410528 # Check if it is already in kernel mode: the kernel
1410529 # has PID 0.
1410530 # If so, just leave (go to L2).
1410531 #
1410532 mov proc_current, %edx # Interrupted PID.
1410533 mov $0, %eax # Kernel PID.
1410534 cmp %eax, %edx
1410535 je L5
1410536 #
1410537 # If we are here, a user process was interrupted.
1410538 # Switch to the kernel stack (data segment
1410539 # descriptor).
1410540 #
1410541 mov $16, %ax
1410542 mov %ax, %ss
1410543 mov _ksp, %esp
1410544 #
1410545 # Call the scheduler.
1410546 #
1410547 call proc_scheduler
1410548 #
1410549 # Restore process stack registers from kernel data
1410550 # segment.
1410551 #
1410552 mov proc_stack_segment_selector, %ss
1410553 mov proc_stack_pointer, %esp
1410554 #
1410555 L5: # Restore from process stack and return.
1410556 #
1410557 jmp irq_pic1_pop_iret
1410558 #-----
1410559 #
1410560 # Generic IRQ from PIC 2
1410561 #
1410562 irq_pic2:
1410563 #
1410564 # Previous pushes:
1410565 # [omissis]
1410566 # push %eflags
1410567 # push %cs
1410568 # push %eip
1410569 #
1410570 pushl %gs
1410571 pushl %fs
1410572 pushl %es
1410573 pushl %ds
1410574 pushl %edi
1410575 pushl %esi
1410576 pushl %ebp
1410577 pushl %ebx
1410578 pushl %edx
1410579 pushl %ecx
1410580 pushl %eax
1410581 #
1410582 # Set the data segments to the kernel data segment,
1410583 # so that the following variables can be accessed.
1410584 #
1410585 mov $16, %ax # DS, ES, FS and GS.
1410586 mov %ax, %ds
1410587 mov %ax, %es
1410588 mov %ax, %fs
1410589 mov %ax, %gs
1410590 #
1410591 # Check if a system call is already working: if so,
1410592 # just leave (go to L2).
1410593 #
1410594 cml $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 cml $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	554
94.7.2	kernel/lib_k/k_gets.c	554
94.7.3	kernel/lib_k/k_perror.c	555
94.7.4	kernel/lib_k/k_printf.c	555
94.7.5	kernel/lib_k/k_sleep.c	555
94.7.6	kernel/lib_k/k_stime.c	556
94.7.7	kernel/lib_k/k_usleep.c	556
94.7.8	kernel/lib_k/k_vprintf.c	557
94.7.9	kernel/lib_k/k_vsprintf.c	557

### 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 (errno != 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.

```

158001 #ifndef _KERNEL_LIB_S_H
158002 #define _KERNEL_LIB_S_H 1
158003 //-----
158004 #include <sys/types.h>
158005 #include <sys/stat.h>
158006 #include <kernel/fs.h>
158007 #include <sys/os32.h>
158008 #include <stddef.h>
158009 #include <stdint.h>
158010 #include <time.h>
158011 #include <termios.h>
158012 #include <setjmp.h>
158013 //-----
158014 void s__exit (pid_t pid, int status);
158015 int s_brk (pid_t pid, void *address);
158016 void *s_sbrk (pid_t pid, intptr_t increment);
158017 pid_t s_fork (pid_t ppid);
158018 int s_kill (pid_t pid_killer, pid_t pid_target, int sig);
158019 void s_longjmp (pid_t pid, jmp_buf env, int val);
158020 int s_seteuid (pid_t pid, uid_t euid);
158021 int s_setjmp (pid_t pid, jmp_buf env);
158022 int s_setuid (pid_t pid, uid_t uid);
158023 int s_setgid (pid_t pid, gid_t egid);
158024 int s_setgid (pid_t pid, gid_t gid);
158025 pid_t s_wait (pid_t pid, int *status);
158026 sighandler_t s_signal (pid_t pid, int sig,
158027                       sighandler_t handler,
158028                       uintptr_t wrapper);
158029 //-----
158030 int s_chdir (pid_t pid, const char *path);
158031 int s_chmod (pid_t pid, const char *path, mode_t mode);
158032 int s_chown (pid_t pid, const char *path, uid_t uid,
158033             gid_t gid);
158034 int s_link (pid_t pid, const char *path_old,
158035            const char *path_new);
158036 int s_mkdir (pid_t pid, const char *path, mode_t mode);
158037 int s_mknod (pid_t pid, const char *path, mode_t mode,
158038             dev_t device);
158039 int s_open (pid_t pid, const char *path, int oflags,
158040            mode_t mode);
158041 int s_stat (pid_t pid, const char *path,
158042            struct stat *buffer);
158043 int s_unlink (pid_t pid, const char *path);
158044 //
158045 int s_pipe (pid_t pid, int pipefd[2]);
158046 //
158047 int s_close (pid_t pid, int fdn);
158048 int s_dup (pid_t pid, int fdn_old);
158049 int s_dup2 (pid_t pid, int fdn_old, int fdn_new);
158050 int s_fchmod (pid_t pid, int fdn, mode_t mode);
158051 int s_fchown (pid_t pid, int fdn, uid_t uid, gid_t gid);
158052 int s_fcntl (pid_t pid, int fdn, int cmd, int arg);
158053 int s_fstat (pid_t pid, int fdn, struct stat *buffer);
158054 off_t s_lseek (pid_t pid, int fdn, off_t offset,
158055              int whence);
158056 ssize_t s_read (pid_t pid, int fdn, void *buffer,
158057               size_t count);
158058 ssize_t s_write (pid_t pid, int fdn,
158059                const void *buffer, size_t count);
158060 //
158061 int s_mount (pid_t pid, const char *path_dev,
158062             const char *path_mnt, int options);
158063 int s_umount (pid_t pid, const char *path_mnt);
158064 //-----
158065 int s_accept (pid_t pid, int sfdn,
158066             struct sockaddr *addr, socklen_t * addrlen);
158067 int s_bind (pid_t pid, int sfdn,
158068            const struct sockaddr *addr, socklen_t addrlen);
158069 int s_connect (pid_t pid, int sfdn,
158070              const struct sockaddr *addr,
158071              socklen_t addrlen);
158072 int s_listen (pid_t pid, int sfdn, int backlog);
158073 int s_socket (pid_t pid, int family, int type,
158074             int protocol);
158075 ssize_t s_send (pid_t pid, int sfdn,
158076               const void *buffer, size_t size, int flags);
158077 ssize_t s_recv (pid_t pid, int sfdn, void *buffer,
158078               size_t size, int flags);
158079 ssize_t s_recvfrom (pid_t pid, int sfdn, void *buffer,
158080                   size_t length, int flags,
158081                   struct sockaddr *addrfrom,
158082                   socklen_t * addrlen);
158083 //
158084 int s_ipconfig (pid_t pid, int n, h_addr_t address, int m);

```

```

158005 int s_routeadd (pid_t pid, h_addr_t destination, int m,
158006               h_addr_t router, int device);
158007 int s_routedel (pid_t pid, h_addr_t destination, int m);
158008 //-----
158009 int s_tcsetattr (pid_t pid, int fdn,
158010               struct termios *termios_p);
158011 int s_tcsetattr (pid_t pid, int fdn, int action,
158012               struct termios *termios_p);
158013 //-----
158014 clock_t s_clock (pid_t pid); // [p]
158015 time_t s_time (pid_t pid, time_t * timer); // [p]
158016 int s_stime (pid_t pid, time_t * timer);
158017 //
158018 // [p] The PID information, here, is not used. The
158019 // argument is required only for syntax coherence
158020 // with other functions, that do
158021 // system calls, inside the kernel.
158022 //-----
158023 #endif

```

94.8.1	kernel/lib_s/s__exit.c	560
94.8.2	kernel/lib_s/s_accept.c	562
94.8.3	kernel/lib_s/s_bind.c	564
94.8.4	kernel/lib_s/s_brk.c	565
94.8.5	kernel/lib_s/s_chdir.c	569
94.8.6	kernel/lib_s/s_chmod.c	569
94.8.7	kernel/lib_s/s_chown.c	570
94.8.8	kernel/lib_s/s_clock.c	571
94.8.9	kernel/lib_s/s_close.c	571
94.8.10	kernel/lib_s/s_connect.c	572
94.8.11	kernel/lib_s/s_dup.c	575
94.8.12	kernel/lib_s/s_dup2.c	575
94.8.13	kernel/lib_s/s_fchmod.c	575
94.8.14	kernel/lib_s/s_fchown.c	576
94.8.15	kernel/lib_s/s_fcntl.c	577
94.8.16	kernel/lib_s/s_fork.c	578
94.8.17	kernel/lib_s/s_fstat.c	582
94.8.18	kernel/lib_s/s_ipconfig.c	583
94.8.19	kernel/lib_s/s_kill.c	584
94.8.20	kernel/lib_s/s_link.c	585
94.8.21	kernel/lib_s/s_listen.c	586
94.8.22	kernel/lib_s/s_longjmp.c	587
94.8.23	kernel/lib_s/s_lseek.c	588
94.8.24	kernel/lib_s/s_mkdir.c	589
94.8.25	kernel/lib_s/s_mknod.c	591
94.8.26	kernel/lib_s/s_mount.c	592
94.8.27	kernel/lib_s/s_open.c	592
94.8.28	kernel/lib_s/s_pipe.c	596
94.8.29	kernel/lib_s/s_read.c	598
94.8.30	kernel/lib_s/s_recvfrom.c	600
94.8.31	kernel/lib_s/s_routeadd.c	606
94.8.32	kernel/lib_s/s_routedel.c	607
94.8.33	kernel/lib_s/s_sbrk.c	608
94.8.34	kernel/lib_s/s_send.c	609
94.8.35	kernel/lib_s/s_setgid.c	612
94.8.36	kernel/lib_s/s_seteuid.c	612
94.8.37	kernel/lib_s/s_setgid.c	613
94.8.38	kernel/lib_s/s_setjmp.c	613

94.8.39	kernel/lib_s/s_setuid.c	614
94.8.40	kernel/lib_s/s_signal.c	614
94.8.41	kernel/lib_s/s_socket.c	615
94.8.42	kernel/lib_s/s_stat.c	617
94.8.43	kernel/lib_s/s_stime.c	618
94.8.44	kernel/lib_s/s_tcgetattr.c	618
94.8.45	kernel/lib_s/s_tcsetattr.c	619
94.8.46	kernel/lib_s/s_time.c	620
94.8.47	kernel/lib_s/s_umount.c	620
94.8.48	kernel/lib_s/s_unlink.c	622
94.8.49	kernel/lib_s/s_wait.c	624
94.8.50	kernel/lib_s/s_write.c	625

## 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;

```

```

162020 addr_t allocated_data;
162021 int status;
162022 //
162023 // All segments start form ((void *) 0), so the new
162024 // address requested is equivalent to the new size
162025 // for the data segment.
162026 //
162027 requested_size = (size_t) address;
162028 //
162029 // Check if it is possible to get the new size:
162030 // cannot be less
162031 // then the original segment size.
162032 //
162033 if (proc_table[pid].domain_data == 0)
162034 {
162035     if (proc_table[pid].domain_text > requested_size)
162036     {
162037         requested_extra = 0;
162038     }
162039     else
162040     {
162041         requested_extra = requested_size
162042             - proc_table[pid].domain_text;
162043     }
162044 }
162045 else
162046 {
162047     if (proc_table[pid].domain_data > requested_size)
162048     {
162049         requested_extra = 0;
162050     }
162051     else
162052     {
162053         requested_extra = requested_size
162054             - proc_table[pid].domain_data;
162055     }
162056 }
162057 //
162058 // Now make shure that the new value is a multiple
162059 // of
162060 // MEM_BLOCK_SIZE!
162061 //
162062 if (requested_extra % MEM_BLOCK_SIZE)
162063 {
162064     requested_extra =
162065         (((requested_extra / MEM_BLOCK_SIZE) +
162066          1) * MEM_BLOCK_SIZE);
162067 }
162068 //
162069 // Now resize the process.
162070 //
162071 if (requested_extra == proc_table[pid].extra_data)
162072 {
162073     //
162074     // Nothing have to change.
162075     //
162076     return (0);
162077 }
162078 else if (requested_extra < proc_table[pid].extra_data)
162079 {
162080     //
162081     // Just reduce.
162082     //
162083     previous_extra = proc_table[pid].extra_data;
162084     proc_table[pid].extra_data = requested_extra;
162085     //
162086     // Update process DATA segment inside the GDT
162087     // table.
162088     //
162089     if (proc_table[pid].domain_data > 0)
162090     {
162091         gdt_segment (gdt_pid_to_segment_data (pid),
162092                     (uint32_t)
162093                     proc_table[pid].address_data,
162094                     (uint32_t) ((proc_table
162095                                 [pid].domain_data +
162096                                 proc_table
162097                                 [pid].extra_data) /
162098                                 MEM_BLOCK_SIZE), 1, 0,
162099                     0);
162100     }
162101     else
162102     {
162103         gdt_segment (gdt_pid_to_segment_data (pid),
162104                     (uint32_t)
162105                     proc_table[pid].address_text,
162106                     (uint32_t) ((proc_table

```

```

162107         [pid].domain_text +
162108         proc_table
162109         [pid].extra_data) /
162110         MEM_BLOCK_SIZE), 1, 0,
162111         0);
162112     }
162113     //
162114     // Release memory.
162115     //
162116     if (proc_table[pid].domain_data > 0)
162117     {
162118         status =
162119             mb_reduce ((proc_table[pid].address_data +
162120                        proc_table[pid].domain_data),
162121                       proc_table[pid].extra_data,
162122                       previous_extra);
162123     }
162124     else
162125     {
162126         status =
162127             mb_reduce ((proc_table[pid].address_text +
162128                        proc_table[pid].domain_text),
162129                       proc_table[pid].extra_data,
162130                       previous_extra);
162131     }
162132     //
162133     if (status < 0)
162134     {
162135         //
162136         // What happened?
162137         //
162138         k_perror (NULL);
162139     }
162140 }
162141 else
162142 {
162143     //
162144     // A bigger size was requested. Save previous
162145     // information.
162146     //
162147     previous_address_text = proc_table[pid].address_text;
162148     previous_domain_text = proc_table[pid].domain_text;
162149     previous_address_data = proc_table[pid].address_data;
162150     previous_domain_data = proc_table[pid].domain_data;
162151     previous_extra = proc_table[pid].extra_data;
162152     //
162153     // Allocate memory for text,
162154     // if text and data are inside the same address
162155     // space;
162156     // otherwise only the data segment is involved.
162157     //
162158     if (proc_table[pid].domain_data == 0)
162159     {
162160         allocated_text =
162161             mb_alloc_size (proc_table[pid].domain_text +
162162                           requested_extra);
162163         //
162164         if (allocated_text == 0)
162165         {
162166             errset (ENOMEM); // Not enough space.
162167             return (-1);
162168         }
162169         //
162170         if (DEBUG)
162171         {
162172             k_printf ("%s:%i:mb_alloc_size(%zi)",
162173                       __FILE__, __LINE__,
162174                       (proc_table[pid].domain_text
162175                        + requested_extra));
162176         }
162177     }
162178     //
162179     // Allocate memory for data, if necessary.
162180     //
162181     if (proc_table[pid].domain_data > 0)
162182     {
162183         allocated_data =
162184             mb_alloc_size (proc_table[pid].domain_data +
162185                           requested_extra);
162186         //
162187         if (allocated_data == 0)
162188         {
162189             //
162190             // Please note that, if we are here, no
162191             // memory
162192             // for the text was allocated!
162193             //

```

```

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 //

```

```

1660042 if (uid != -1)
1660043 {
1660044 inode->uid = uid;
1660045 inode->changed = 1;
1660046 }
1660047 if (gid != -1)
1660048 {
1660049 inode->gid = gid;
1660050 inode->changed = 1;
1660051 }
1660052 //
1660053 // Save and release the inode.
1660054 //
1660055 inode_save (inode);
1660056 inode_put (inode);
1660057 //
1660058 // Return.
1660059 //
1660060 return (0);
1660061 }

```

#### 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;

```

```

1720011 //
1720012 // Get process.
1720013 //
1720014 ps = proc_reference (pid);
1720015 //
1720016 // Verify if the file descriptor is valid.
1720017 //
1720018 if (ps->fd[fdn].file == NULL)
1720019 {
1720020     errset (EBADF); // Bad file descriptor.
1720021     return (-1);
1720022 }
1720023 //
1720024 // Reach the inode.
1720025 //
1720026 inode = ps->fd[fdn].file->inode;
1720027 //
1720028 // If the Inode does not exist, exit with error.
1720029 //
1720030 if (inode == NULL)
1720031 {
1720032     errset (ENOENT);
1720033     return (-1);
1720034 }
1720035 //
1720036 // Verify to be the owner, or at least to be UID ==
1720037 // 0.
1720038 //
1720039 if (ps->euid != inode->uid && ps->euid != 0)
1720040 {
1720041     errset (EACCES); // Permission denied.
1720042     return (-1);
1720043 }
1720044 //
1720045 // Update the mode: the file type is kept and the
1720046 // rest is taken form the parameter 'mode'.
1720047 //
1720048 inode->mode = (S_IFMT & inode->mode) | (~S_IFMT & mode);
1720049 //
1720050 // Save the inode.
1720051 //
1720052 inode->changed = 1;
1720053 inode_save (inode);
1720054 //
1720055 // Return.
1720056 //
1720057 return (0);
1720058 }

```

#### 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.

```

1720001 #include <kernel/proc.h>
1720002 #include <kernel/lib_s.h>
1720003 #include <kernel/fs.h>
1720004 #include <errno.h>
1720005 #include <fcntl.h>
1720006 //-----
1720007 int
1720008 s_fcntl (pid_t pid, int fdn, int cmd, int arg)
1720009 {
1720010     proc_t *ps;
1720011     int mask;
1720012 //
1720013 // Get process.
1720014 //
1720015 ps = proc_reference (pid);
1720016 //
1720017 // Verify if the file descriptor is valid.
1720018 //
1720019 if (ps->fd[fdn].file == NULL)
1720020 {
1720021     errset (EBADF); // Bad file descriptor.
1720022     return (-1);
1720023 }
1720024 //
1720025 //
1720026 //
1720027 switch (cmd)
1720028 {
1720029     case F_DUPFD:
1720030         return (fd_dup (pid, fdn, arg));
1720031         break;
1720032     case F_GETFD:
1720033         return (ps->fd[fdn].fd_flags);
1720034         break;
1720035     case F_SETFD:
1720036         ps->fd[fdn].fd_flags = arg;
1720037         return (0);
1720038     case F_GETFL:
1720039         return (ps->fd[fdn].fl_flags);
1720040     case F_SETFL:
1720041         // Calculate a mask with bits that are *not* to
1720042         // be set.
1720043         //
1720044         mask =
1720045             (O_ACCMODE | O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);
1720046         //
1720047         // Set to zero the bits that are not to be set
1720048         // from
1720049         // the argument.
1720050         //

```

```

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 }

```

```

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 }

```

## 94.8.17 kernel/lib\_s/s\_fstat.c

&lt;&lt;

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

```

## 94.8.18 kernel/lib\_s/s\_ipconfig.c

&lt;&lt;

Si veda la sezione 87.28.

```

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

```

```
1760066 }

```

## 94.8.19 kernel/lib\_s/s\_kill.c

« Si veda la sezione 87.29.

```
1770001 #include <kernel/proc.h>
1770002 #include <kernel/lib_s.h>
1770003 #include <errno.h>
1770004 //-----
1770005 int
1770006 s_kill (pid_t pid_killer, pid_t pid_target, int sig)
1770007 {
1770008     uid_t euid = proc_table[pid_killer].euid;
1770009     uid_t uid = proc_table[pid_killer].uid;
1770010     pid_t pgrp = proc_table[pid_killer].pgrp;
1770011     int p; // Index inside the process table.
1770012     //
1770013     if (pid_target < -1)
1770014     {
1770015         errset (ESRCH);
1770016         return (-1);
1770017     }
1770018     else if (pid_target == -1)
1770019     {
1770020         if (sig == 0)
1770021         {
1770022             return (0);
1770023         }
1770024         if (euid == 0)
1770025         {
1770026             //
1770027             // Because 'pid_target' is equal to '-1' and
1770028             // the effective
1770029             // user identity is '0', then, all
1770030             // processes,
1770031             // except the kernel and init, will receive
1770032             // the signal.
1770033             //
1770034             // The following scan starts from 2, to
1770035             // preserve the
1770036             // kernel and init processes.
1770037             //
1770038             for (p = 2; p < PROCESS_MAX; p++)
1770039             {
1770040                 if (proc_table[p].status != PROC_EMPTY
1770041                     && proc_table[p].status != PROC_ZOMBIE)
1770042                 {
1770043                     proc_sig_on (p, sig);
1770044                 }
1770045             }
1770046         }
1770047         else
1770048         {
1770049             //
1770050             // Because 'pid_target' is equal to '-1', but
1770051             // the effective
1770052             // user identity is not '0', then, all
1770053             // processes owned
1770054             // by the same effective user identity, will
1770055             // receive the
1770056             // signal.
1770057             //
1770058             // The following scan starts from 1, to
1770059             // preserve the
1770060             // kernel process.
1770061             //
1770062             for (p = 1; p < PROCESS_MAX; p++)
1770063             {
1770064                 if (proc_table[p].status != PROC_EMPTY
1770065                     && proc_table[p].status !=
1770066                     PROC_ZOMBIE && proc_table[p].uid == euid)
1770067                 {
1770068                     proc_sig_on (p, sig);
1770069                 }
1770070             }
1770071             return (0);
1770072         }
1770073     }
1770074     else if (pid_target == 0)
1770075     {
1770076         if (sig == 0)
1770077         {
1770078             return (0);
1770079         }
1770080         //
1770081         // The following scan starts from 1, to preserve

```

```
1770082 // the
1770083 // kernel process.
1770084 //
1770085     for (p = 1; p < PROCESS_MAX; p++)
1770086     {
1770087         if (proc_table[p].status != PROC_EMPTY
1770088             && proc_table[p].status != PROC_ZOMBIE
1770089             && proc_table[p].pgrp == pgrp)
1770090         {
1770091             proc_sig_on (p, sig);
1770092         }
1770093     }
1770094     return (0);
1770095 }
1770096 else if (pid_target >= PROCESS_MAX)
1770097 {
1770098     errset (ESRCH);
1770099     return (-1);
1770100 }
1770101 else // (pid_target > 0)
1770102 {
1770103     if (proc_table[pid_target].status == PROC_EMPTY
1770104         || proc_table[pid_target].status == PROC_ZOMBIE)
1770105     {
1770106         errset (ESRCH);
1770107         return (-1);
1770108     }
1770109     else if (uid == proc_table[pid_target].uid ||
1770110             uid == proc_table[pid_target].suid ||
1770111             euid == proc_table[pid_target].uid ||
1770112             euid == proc_table[pid_target].suid
1770113             || euid == 0)
1770114     {
1770115         if (sig == 0)
1770116         {
1770117             return (0);
1770118         }
1770119         else
1770120         {
1770121             proc_sig_on (pid_target, sig);
1770122             return (0);
1770123         }
1770124     }
1770125     else
1770126     {
1770127         errset (EPERM);
1770128         return (-1);
1770129     }
1770130 }
1770131 }
```

## 94.8.20 kernel/lib\_s/s\_link.c

« Si veda la sezione 87.30.

```
1780001 #include <kernel/fs.h>
1780002 #include <errno.h>
1780003 #include <kernel/proc.h>
1780004 #include <kernel/lib_s.h>
1780005 //-----
1780006 int
1780007 s_link (pid_t pid, const char *path_old,
1780008         const char *path_new)
1780009 {
1780010     proc_t *ps;
1780011     inode_t *inode_old;
1780012     inode_t *inode_new;
1780013     char path_new_full[PATH_MAX];
1780014     //
1780015     // Get process.
1780016     //
1780017     ps = proc_reference (pid);
1780018     //
1780019     // Try to get the old path inode.
1780020     //
1780021     inode_old = path_inode (pid, path_old);
1780022     if (inode_old == NULL)
1780023     {
1780024         //
1780025         // Cannot get the inode: 'errno' is already set
1780026         // by
1780027         // 'path_inode()'.
1780028         //
1780029         errset (errno);
1780030         return (-1);
1780031     }
1780032     //

```

```

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

```

#### 94.8.21 kernel/lib\_s/s\_listen.c

Si veda la sezione 87.31.

```

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

```

```

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

```

#### 94.8.22 kernel/lib\_s/s\_longjmp.c

Si veda la sezione 87.49.

```

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

```



```

180035 sp->ebx0 = jmpenv->ebx0;
180036 sp->ebp0 = jmpenv->ebp0;
180037 sp->esi0 = jmpenv->esi0;
180038 sp->edi0 = jmpenv->edi0;
180039 sp->ds0 = jmpenv->ds0;
180040 sp->es0 = jmpenv->es0;
180041 sp->fs0 = jmpenv->fs0;
180042 sp->gs0 = jmpenv->gs0;
180043 sp->eflags0 = jmpenv->eflags0;
180044 sp->cs0 = jmpenv->cs0;
180045 sp->eip0 = jmpenv->eip0;
180046 //
180047 sp->eipl = jmpenv->eipl;
180048 sp->syscallnr = jmpenv->syscallnr;
180049 sp->msg_pointer = jmpenv->msg_pointer;
180050 sp->msg_size = jmpenv->msg_size;
180051 sp->env = jmpenv->env;
180052 sp->ret = val;
180053 sp->ebp1 = jmpenv->ebp1;
180054 sp->eip2 = jmpenv->eip2;
180055 //
180056 // Replace the stack pointer too!
180057 //
180058 proc_stack_pointer = jmpenv->esp0;
180059 }

```

## 94.8.23 kernel/lib\_s/s\_lseek.c

« Si veda la sezione 87.33.

```

181001 #include <kernel/proc.h>
181002 #include <kernel/lib_s.h>
181003 #include <errno.h>
181004 //-----
181005 off_t
181006 s_lseek (pid_t pid, int fdn, off_t offset, int whence)
181007 {
181008     inode_t *inode;
181009     file_t *file;
181010     fd_t *fd;
181011     off_t test_offset;
181012 //
181013 // Get file descriptor.
181014 //
181015 fd = fd_reference (pid, &fdn);
181016 if (fd == NULL || fd->file == NULL
181017     || fd->file->inode == NULL)
181018     {
181019         errset (EBADF); // Bad file descriptor.
181020         return (-1);
181021     }
181022 //
181023 // Get file table item.
181024 //
181025 file = fd->file;
181026 //
181027 // Get inode.
181028 //
181029 inode = file->inode;
181030 //
181031 // Change position depending on the 'whence'
181032 // parameter.
181033 //
181034 if (whence == SEEK_SET)
181035     {
181036         if (offset < 0)
181037             {
181038                 errset (EINVAL); // Invalid argument.
181039                 return ((off_t) - 1);
181040             }
181041         else
181042             {
181043                 fd->file->offset = offset;
181044             }
181045     }
181046 else if (whence == SEEK_CUR)
181047     {
181048         test_offset = fd->file->offset;
181049         test_offset += offset;
181050         if (test_offset < 0)
181051             {
181052                 errset (EINVAL); // Invalid argument.
181053                 return ((off_t) - 1);
181054             }
181055         else
181056             {
181057                 fd->file->offset = test_offset;

```

```

181058     }
181059 }
181060 else if (whence == SEEK_END)
181061     {
181062         test_offset = inode->size;
181063         test_offset += offset;
181064         if (test_offset < 0)
181065             {
181066                 errset (EINVAL); // Invalid argument.
181067                 return ((off_t) - 1);
181068             }
181069         else
181070             {
181071                 fd->file->offset = test_offset;
181072             }
181073     }
181074 else
181075     {
181076         errset (EINVAL); // Invalid argument.
181077         return ((off_t) - 1);
181078     }
181079 //
181080 // Return the new file position.
181081 //
181082 return (fd->file->offset);
181083 }

```

## 94.8.24 kernel/lib\_s/s\_mkdir.c

« Si veda la sezione 87.34.

```

182001 #include <kernel/fs.h>
182002 #include <errno.h>
182003 #include <kernel/proc.h>
182004 #include <libgen.h>
182005 #include <kernel/lib_s.h>
182006 //-----
182007 int
182008 s_mkdir (pid_t pid, const char *path, mode_t mode)
182009 {
182010     proc_t *ps;
182011     inode_t *inode_directory;
182012     inode_t *inode_parent;
182013     int status;
182014     char path_directory[PATH_MAX];
182015     char path_copy[PATH_MAX];
182016     char *path_parent;
182017     ssize_t size_written;
182018 //
182019 struct
182020     {
182021         ino_t ino_1;
182022         char name_1[NAME_MAX];
182023         ino_t ino_2;
182024         char name_2[NAME_MAX];
182025     } directory;
182026 //
182027 // Get process.
182028 //
182029 ps = proc_reference (pid);
182030 //
182031 // Correct the mode with the umask.
182032 //
182033 mode &= ~ps->umask;
182034 //
182035 // Inside 'mode', the file type is fixed. No check
182036 // is made.
182037 //
182038 mode &= 00777;
182039 mode |= S_IFDIR;
182040 //
182041 // The full path and the directory path is needed.
182042 //
182043 status = path_full (path, ps->path_cwd, path_directory);
182044 if (status < 0)
182045     {
182046         return (-1);
182047     }
182048 strncpy (path_copy, path_directory, PATH_MAX);
182049 path_copy[PATH_MAX - 1] = 0;
182050 path_parent = dirname (path_copy);
182051 //
182052 // Check if something already exists with the same
182053 // name. The scan
182054 // is done with kernel privileges.
182055 //
182056 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 create 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 | 0111;
1850040     mode &= ~umask;
1850041     //
1850042     // Check open options.
1850043     //
1850044     if (oflags & O_WRONLY)
1850045     {
1850046         //
1850047         // The file is to be opened for write, or for
1850048         // read/write.
1850049         // Try to get inode.
1850050         //
1850051         inode = path_inode (pid, path);
1850052         if (inode == NULL)
1850053         {
1850054             //
1850055             // Cannot get the inode. See if there is the
1850056             // creation
1850057             // option.
1850058             //
1850059             if (oflags & O_CREAT)
1850060             {
1850061                 //
1850062                 // Try to create the missing inode: the
1850063                 // file must be a
1850064                 // regular one, so add the mode.
1850065                 //
1850066                 path_full (path,
1850067                            proc_table[pid].path_cwd,
1850068                            full_path);
1850069                 inode =
1850070                     path_inode_link (pid, full_path, NULL,
1850071                                     (mode | S_IFREG));
1850072                 if (inode == NULL)
1850073                 {
1850074                     //
1850075                     // Sorry: cannot create the inode!
1850076                     // Variable 'errno'
1850077                     // is already set by
1850078                     // 'path_inode_link()'.
1850079                     //
1850080                     errset (errno);
1850081                     return (-1);
1850082                 }
1850083             }
1850084             else
1850085             {
1850086                 //
1850087                 // Cannot open the inode. Variable
1850088                 // 'errno'
1850089                 // should be already set by
1850090                 // 'path_inode()'.
1850091                 //
1850092                 errset (errno);
1850093                 return (-1);
1850094             }
1850095         }
1850096         //
1850097         // The inode was read or created: check if it
1850098         // must be
1850099         // truncated. It can be truncated only if it is

```

```

1850100 // a regular
1850101 // file.
1850102 //
1850103 if (oflags & O_TRUNC && inode->mode & S_IFREG)
1850104 {
1850105 //
1850106 // Truncate inode.
1850107 //
1850108 status = inode_truncate (inode);
1850109 if (status != 0)
1850110 {
1850111 //
1850112 // Cannot truncate the inode: release it
1850113 // and return.
1850114 // But this error should never happen,
1850115 // because the
1850116 // function 'inode_truncate()' will not
1850117 // return any
1850118 // other value than zero.
1850119 //
1850120 errno_save = errno;
1850121 inode_put (inode);
1850122 errset (errno_save);
1850123 return (-1);
1850124 }
1850125 }
1850126 }
1850127 else
1850128 {
1850129 //
1850130 // The file is to be opened for read, but not
1850131 // for write.
1850132 // Try to get inode.
1850133 //
1850134 inode = path_inode (pid, path);
1850135 if (inode == NULL)
1850136 {
1850137 //
1850138 // Cannot open the file.
1850139 //
1850140 errset (errno);
1850141 return (-1);
1850142 }
1850143 }
1850144 //
1850145 // An inode was opened: check type and access
1850146 // permissions.
1850147 // All file types are good, even directories, as the
1850148 // type
1850149 // DIR is implemented through file descriptors.
1850150 //
1850151 perm = 0;
1850152 if (oflags & O_RDONLY)
1850153 perm |= 4;
1850154 if (oflags & O_WRONLY)
1850155 perm |= 2;
1850156 status =
1850157 inode_check (inode, S_IFMT, perm,
1850158 proc_table[pid].euid,
1850159 proc_table[pid].egid);
1850160 if (status != 0)
1850161 {
1850162 //
1850163 // The file type is not correct or the user does
1850164 // not have
1850165 // permissions.
1850166 //
1850167 return (-1);
1850168 }
1850169 //
1850170 // Allocate the file, inside the file table.
1850171 //
1850172 file = file_reference (-1);
1850173 if (file == NULL)
1850174 {
1850175 //
1850176 // Cannot allocate the file inside the file
1850177 // table: release the
1850178 // inode, update 'errno' and return.
1850179 //
1850180 inode_put (inode);
1850181 errset (ENFILE); // Too many files open in
1850182 // system.
1850183 return (-1);
1850184 }
1850185 //
1850186 // Put some data inside the file item. Only options

```

```

1850187 // O_RDONLY and O_WRONLY are kept here, because the
1850188 // O_APPEND
1850189 // is saved inside the file descriptor table.
1850190 //
1850191 file->references = 1;
1850192 file->oflags = (oflags & (O_RDONLY | O_WRONLY));
1850193 file->inode = inode;
1850194 file->sock = NULL;
1850195 //
1850196 // Allocate the file descriptor: variable 'fdn' will
1850197 // be modified
1850198 // by the call to 'fd_reference()'.
1850199 //
1850200 fdn = -1;
1850201 fd = fd_reference (pid, &fdn);
1850202 if (fd == NULL)
1850203 {
1850204 //
1850205 // Cannot allocate the file descriptor: remove
1850206 // the item from
1850207 // file table.
1850208 //
1850209 file->references = 0;
1850210 file->oflags = 0;
1850211 file->inode = NULL;
1850212 file->sock = NULL;
1850213 //
1850214 // Release the inode.
1850215 //
1850216 inode_put (inode);
1850217 //
1850218 // Return an error.
1850219 //
1850220 errset (EMFILE); // Too many open files.
1850221 return (-1);
1850222 }
1850223 //
1850224 // File descriptor allocated: put some data inside
1850225 // the
1850226 // file descriptor item.
1850227 //
1850228 fd->fl_flags =
1850229 (oflags & (O_RDONLY | O_WRONLY | O_APPEND));
1850230 fd->fd_flags = 0;
1850231 fd->file = file;
1850232 fd->file->offset = 0;
1850233 //
1850234 // Check for particular types and situations.
1850235 //
1850236 if ((S_ISCHR (inode->mode))
1850237 && (oflags & O_RDONLY) && (oflags & O_WRONLY))
1850238 {
1850239 //
1850240 // The inode is a character special file
1850241 // (related to a character
1850242 // device), opened for read and write!
1850243 //
1850244 if ((inode->direct[0] & 0xFF00) ==
1850245 (DEV_CONSOLE_MAJOR << 8))
1850246 {
1850247 //
1850248 // It is a terminal (currently only consoles
1850249 // are possible).
1850250 // Get the tty reference.
1850251 //
1850252 tty = tty_reference ((dev_t) inode->direct[0]);
1850253 //
1850254 // Verify that the terminal is not already
1850255 // the controlling
1850256 // terminal of some process group.
1850257 //
1850258 if (tty->pgrp == 0)
1850259 {
1850260 //
1850261 // The terminal is free: verify if the
1850262 // current process
1850263 // needs a controlling terminal.
1850264 //
1850265 if (proc_table[pid].device_tty == 0
1850266 && proc_table[pid].pgrp == pid)
1850267 {
1850268 //
1850269 // It is a group leader with no
1850270 // controlling
1850271 // terminal: set the controlling
1850272 // terminal.
1850273 //

```

```

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

```

#### 94.8.28 kernel/lib\_s/s\_pipe.c

◀ Si veda la sezione 87.38.

```

1860001 #include <kernel/proc.h>
1860002 #include <kernel/lib_s.h>
1860003 #include <kernel/lib_k.h>
1860004 #include <errno.h>
1860005 #include <fcntl.h>
1860006 //-----
1860007 int
1860008 s_pipe (pid_t pid, int pipefd[2])
1860009 {
1860010     file_t *file;
1860011     fd_t *fd_read;
1860012     fd_t *fd_write;
1860013     int fdn_read;
1860014     int fdn_write;
1860015     //
1860016     // Allocate the file inside the file table and the
1860017     // inode inside

```

```

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

```

```

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

```

## 94.8.29 kernel/lib\_s/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     }

```



```

1880131 //
1880132 ;
1880133 }
1880134 else
1880135 {
1880136     continue;
1880137 }
1880138 }
1880139 //
1880140 // Packet accepted.
1880141 //
1880142 // This ICMP RAW packet is new for
1880143 // the
1880144 // socket: save the clock time, so
1880145 // that the
1880146 // same packet is not read again.
1880147 //
1880148 sfd->file->sock->read.clock[i]
1880149     = ip_table[i].clock;
1880150 //
1880151 // Copy the packet.
1880152 //
1880153 size_read
1880154     =
1880155     min (ntohs
1880156         (ip_table[i].packet.header.
1880157          tot_len), length);
1880158 //
1880159 memcpy (buffer,
1880160         ip_table[i].packet.octet,
1880161         size_read);
1880162 //
1880163 // Get the source address and
1880164 // return.
1880165 //
1880166 if (addrfrom != NULL && addrlen != NULL)
1880167     {
1880168         if (*addrlen >=
1880169             sizeof (struct sockaddr_in))
1880170             {
1880171                 addrfrom_in->sin_family = AF_INET;
1880172                 addrfrom_in->sin_port = 0;
1880173                 addrfrom_in->sin_addr
1880174                     =
1880175                     ip_table[i].packet.header.saddr;
1880176             }
1880177             *addrlen =
1880178                 sizeof (struct sockaddr_in);
1880179         }
1880180     return ((ssize_t) size_read);
1880181 }
1880182 }
1880183 else
1880184 {
1880185     //
1880186     // Unsupported protocol.
1880187     //
1880188     errset (EPROTONOSUPPORT);
1880189     return ((ssize_t) - 1);
1880190 }
1880191 }
1880192 else if (sfd->file->sock->type == SOCK_DGRAM)
1880193     {
1880194         //
1880195         // DGRAM
1880196         //
1880197         if (sfd->file->sock->protocol == IPPROTO_UDP)
1880198             {
1880199                 //
1880200                 // UDP
1880201                 //
1880202                 // Scan the ip_table[] to find an UDP
1880203                 // packet
1880204                 // that was not already seen by the
1880205                 // socket.
1880206                 //
1880207                 for (i = 0; i < IP_MAX_PACKETS; i++)
1880208                     {
1880209                         //
1880210                         // Check the protocol.
1880211                         //
1880212                         if (ip_table[i].packet.header.protocol !=
1880213                             IPPROTO_UDP)
1880214                             {
1880215                                 //
1880216                                 // It is not UDP.
1880217                                 //

```

```

1880218         continue;
1880219     }
1880220 }
1880221 //
1880222 // Is the packet new for the socket?
1880223 //
1880224 // Please notice that the kernel
1880225 // might be interrupted
1880226 // also between clock tics; so,
1880227 // during a single clock
1880228 // time, a new packet might be
1880229 // reached.
1880230 //
1880231 if (ip_table[i].clock
1880232     < sfd->file->sock->read.clock[i])
1880233     {
1880234         //
1880235         // Already seen or packet too
1880236         // old.
1880237         //
1880238         continue;
1880239     }
1880240 //
1880241 // Verify the ports.
1880242 //
1880243 udp = (struct udphdr *)
1880244     &ip_table[i].packet.octet
1880245     [sizeof (struct iphdr)];
1880246 //
1880247 if (udp->dest == 0)
1880248     {
1880249         //
1880250         // Cannot accept packets for the
1880251         // port zero!
1880252         //
1880253         continue;
1880254     }
1880255 //
1880256 if (udp->dest !=
1880257     htons (sfd->file->sock->lport))
1880258     {
1880259         //
1880260         // The local port does not
1880261         // match!
1880262         //
1880263         continue;
1880264     }
1880265 //
1880266 if (udp->source !=
1880267     htons (sfd->file->sock->rport)
1880268     && sfd->file->sock->rport != 0)
1880269     {
1880270         //
1880271         // The remote port does not
1880272         // match, and is not
1880273         // zero.
1880274         //
1880275         continue;
1880276     }
1880277 //
1880278 // Verify the IP addresses.
1880279 //
1880280 if (ip_table[i].packet.header.daddr
1880281     != htonl (sfd->file->sock->laddr)
1880282     && sfd->file->sock->laddr != 0)
1880283     {
1880284         //
1880285         // The local address does not
1880286         // match, and is
1880287         // not zero.
1880288         //
1880289         continue;
1880290     }
1880291 //
1880292 if (ip_table[i].packet.header.saddr
1880293     != htonl (sfd->file->sock->raddr)
1880294     && sfd->file->sock->raddr != 0)
1880295     {
1880296         //
1880297         // The remote address does not
1880298         // match, and is
1880299         // not zero.
1880300         //
1880301         continue;
1880302     }
1880303 //
1880304 // The packet is accepted.
1880305 //

```



```

1880305 // This UDP packet is new for the
1880306 // socket:
1880307 // save the clock time, so that the
1880308 // same packet is not read again.
1880309 //
1880310 sfd->file->sock->read_clock[i]
1880311 = ip_table[i].clock;
1880312 //
1880313 // Check the right minimal size to
1880314 // be read, comparing
1880315 // the size of the IP packet, the
1880316 // size of the UDP
1880317 // packet and the size requested.
1880318 //
1880319 size_read =
1880320 ntohs (ip_table[i].packet.header.
1880321 tot_len) -
1880322 (ip_table[i].packet.header.ihl * 4) -
1880323 (sizeof (struct udphdr));
1880324 size_read =
1880325 min (size_read,
1880326 (udp->len -
1880327 sizeof (struct udphdr)));
1880328 size_read = min (size_read, length);
1880329 //
1880330 // Copy the data inside the UDP
1880331 // packet.
1880332 //
1880333 data =
1880334 ((uint8_t *) udp) +
1880335 sizeof (struct udphdr);
1880336 //
1880337 memcpy (buffer, data, size_read);
1880338 //
1880339 // Get the source address and
1880340 // return.
1880341 //
1880342 if (addrfrom != NULL && addrrlen != NULL)
1880343 {
1880344     if (*addrrlen >=
1880345         sizeof (struct sockaddr_in))
1880346     {
1880347         addrfrom_in->sin_family = AF_INET;
1880348         addrfrom_in->sin_port =
1880349             udp->source;
1880350         addrfrom_in->sin_addr.s_addr =
1880351             ip_table[i].packet.header.saddr;
1880352     }
1880353     *addrrlen =
1880354         sizeof (struct sockaddr_in);
1880355 }
1880356 return ((ssize_t) size_read);
1880357 }
1880358 }
1880359 }
1880360 else if (sfd->file->sock->type == SOCK_STREAM)
1880361 {
1880362     //
1880363     // STREAM
1880364     //
1880365     if (sfd->file->sock->protocol == IPPROTO_TCP)
1880366     {
1880367         //
1880368         // TCP
1880369         //
1880370         // See if the read side of the stream
1880371         // was closed.
1880372         //
1880373         if (sfd->file->sock->tcp.recv_closed
1880374             || sfd->file->sock->tcp.conn == TCP_CLOSE)
1880375         {
1880376             //
1880377             // If the 'recv_size' is zero, the
1880378             // stream
1880379             // is closed.
1880380             //
1880381             if (sfd->file->sock->tcp.recv_size
1880382                 == 0
1880383                 || sfd->file->sock->tcp.can_read == 0)
1880384             {
1880385                 //
1880386                 // End of file.
1880387                 //
1880388                 return ((ssize_t) 0);
1880389             }
1880390         }
1880391     }

```

```

1880392 // At the moment, nothing was read.
1880393 //
1880394 size_read = 0;
1880395 //
1880396 // See if there is data to be read from
1880397 // the stream.
1880398 //
1880399 if (sfd->file->sock->tcp.can_read)
1880400 {
1880401     size_read =
1880402         min (sfd->file->sock->tcp.recv_size,
1880403             length);
1880404     memcpy (buffer,
1880405             sfd->file->sock->tcp.recv_index,
1880406             size_read);
1880407     //
1880408     sfd->file->sock->tcp.recv_size -=
1880409         size_read;
1880410     sfd->file->sock->tcp.recv_index +=
1880411         size_read;
1880412     //
1880413     if (sfd->file->sock->tcp.recv_size == 0)
1880414     {
1880415         //
1880416         // Nothing to be read at the
1880417         // moment.
1880418         //
1880419         sfd->file->sock->tcp.can_read = 0;
1880420         sfd->file->sock->tcp.can_recv = 1;
1880421     }
1880422     //
1880423     // Get the source address and
1880424     // return.
1880425     //
1880426     if (addrfrom != NULL && addrrlen != NULL)
1880427     {
1880428         if (*addrrlen >=
1880429             sizeof (struct sockaddr_in))
1880430         {
1880431             addrfrom_in->sin_family = AF_INET;
1880432             addrfrom_in->sin_port =
1880433                 htons (sfd->file->sock->rport);
1880434             addrfrom_in->sin_addr.s_addr =
1880435                 htons (sfd->file->sock->raddr);
1880436         }
1880437         *addrrlen =
1880438             sizeof (struct sockaddr_in);
1880439     }
1880440 }
1880441 //
1880442 // Check if something was read.
1880443 //
1880444 if (size_read > 0)
1880445 {
1880446     //
1880447     // Return normally.
1880448     //
1880449     return ((ssize_t) size_read);
1880450 }
1880451 else
1880452 {
1880453     //
1880454     // Nothing to be read at the moment.
1880455     //
1880456     if (sfd->fl_flags & O_NONBLOCK)
1880457     {
1880458         //
1880459         // Try again.
1880460         //
1880461         errset (EAGAIN);
1880462         return ((ssize_t) - 1);
1880463     }
1880464     else
1880465     {
1880466         //
1880467         // Go to sleep and return a
1880468         // temporary error.
1880469         //
1880470         proc_table[pid].status =
1880471             PROC_SLEEPING;
1880472         proc_table[pid].ret = 0;
1880473         proc_table[pid].wakeup_events
1880474             = WAKEUP_EVENT_SOCKET_READ;
1880475         proc_table[pid].wakeup_sock =
1880476             sfd->file->sock;
1880477         if (DEBUG)
1880478         {

```

```

1880479         k_printf
1880480             ("[%s:%i] PID %i goes to "
1880481              "sleep waiting to "
1880482              "receive for a socket.\n",
1880483              __FILE__, __LINE__, pid);
1880484     }
1880485     //
1880486     // Nothing was received.
1880487     //
1880488     errset (EAGAIN);
1880489     return ((ssize_t) - 1);
1880490 }
1880491 }
1880492 }
1880493 else
1880494 {
1880495     //
1880496     // Unsupported protocol.
1880497     //
1880498     errset (EPROTONOSUPPORT);
1880499     return ((ssize_t) - 1);
1880500 }
1880501 }
1880502 else
1880503 {
1880504     //
1880505     // Unsupported type.
1880506     //
1880507     errset (EPROTONOSUPPORT);
1880508     return ((ssize_t) - 1);
1880509 }
1880510 }
1880511 else
1880512 {
1880513     //
1880514     // Unsupported family.
1880515     //
1880516     errset (EAFNOSUPPORT);
1880517     return ((ssize_t) - 1);
1880518 }
1880519 //
1880520 // If we are here, there are no more packets to read
1880521 // at the moment.
1880522 //
1880523 if (sfd->fl_flags & O_NONBLOCK)
1880524 {
1880525     //
1880526     // Try again.
1880527     //
1880528     errset (EAGAIN);
1880529     return (-1);
1880530 }
1880531 else
1880532 {
1880533     //
1880534     // The process should go to sleep.
1880535     //
1880536     proc_table[pid].status = PROC_SLEEPING;
1880537     proc_table[pid].ret = 0;
1880538     proc_table[pid].wakeup_events =
1880539         WAKEUP_EVENT_SOCKET_READ;
1880540     proc_table[pid].wakeup_sock = sfd->file->sock;
1880541     if (DEBUG)
1880542     {
1880543         k_printf ("[%s:%i] PID %i goes to sleep "
1880544                  "waiting to receive "
1880545                  "for a socket.\n",
1880546                  __FILE__, __LINE__, pid);
1880547     }
1880548     //
1880549     // Try again.
1880550     //
1880551     errset (EAGAIN);
1880552     return ((ssize_t) - 1);
1880553 }
1880554 }

```

### 94.8.31 kernel/lib\_s/s\_routead.c

Si veda la sezione 87.42.

```

1890001 #include <arpa/inet.h>
1890002 #include <sys/os32.h>
1890003 #include <kernel/net/route.h>
1890004 #include <kernel/lib_k.h>
1890005 #include <errno.h>
1890006 #include <netinet/in.h>

```

```

1890007 #include <kernel/proc.h>
1890008 //-----
1890009 // This syscall is present only inside os32.
1890010 //-----
1890011 int
1890012 s_routead (pid_t pid, in_addr_t dest, int m,
1890013           in_addr_t router, int device)
1890014 {
1890015     int r;
1890016     h_addr_t netmask;
1890017     h_addr_t network;
1890018     //
1890019     // Must be a privileged process.
1890020     //
1890021     if (proc_table[pid].euid != 0)
1890022     {
1890023         errset (EPERM);
1890024         return (-1);
1890025     }
1890026     //
1890027     //
1890028     //
1890029     if (m > 32 || m < 0)
1890030     {
1890031         errset (EINVAL);
1890032         return (-1);
1890033     }
1890034     //
1890035     // Calculate the netmask.
1890036     //
1890037     netmask = ip_mask (m);
1890038     //
1890039     // Fix the destination address, with the mask.
1890040     //
1890041     network = ntohl (dest) & netmask;
1890042     //
1890043     // Check if there is already. If there is: update
1890044     // it.
1890045     //
1890046     for (r = 0; r < ROUTE_MAX_ROUTES; r++)
1890047     {
1890048         if (network == route_table[r].network
1890049             && m == route_table[r].m)
1890050         {
1890051             //
1890052             // Update.
1890053             //
1890054             route_table[r].router = ntohl (router);
1890055             route_table[r].netmask = netmask;
1890056             route_table[r].interface = device;
1890057             return (0);
1890058         }
1890059     }
1890060     //
1890061     // The item is new. Find an empty place.
1890062     //
1890063     for (r = 0; r < ROUTE_MAX_ROUTES; r++)
1890064     {
1890065         if (route_table[r].network == 0xFFFFFFFF)
1890066         {
1890067             //
1890068             // Empty.
1890069             //
1890070             route_table[r].network = network;
1890071             route_table[r].netmask = netmask;
1890072             route_table[r].m = m;
1890073             route_table[r].router = ntohl (router);
1890074             route_table[r].interface = device;
1890075             //
1890076             route_sort ();
1890077             //
1890078             return (0);
1890079         }
1890080     }
1890081     //
1890082     // No free space found.
1890083     //
1890084     errset (ENOMEM);
1890085     return (-1);
1890086 }

```

### 94.8.32 kernel/lib\_s/s\_routedel.c

Si veda la sezione 87.43.

```

1900001 #include <arpa/inet.h>
1900002 #include <sys/os32.h>

```

```

190003 #include <kernel/net/route.h>
190004 #include <kernel/lib_k.h>
190005 #include <errno.h>
190006 #include <netinet/in.h>
190007 #include <kernel/proc.h>
190008 //-----
190009 // This syscall is present only inside os32.
190010 //-----
190011 int
190012 s_routedel (pid_t pid, in_addr_t dest, int m)
190013 {
190014     int r;
190015     h_addr_t network;
190016     //
190017     // Must be a privileged process.
190018     //
190019     if (proc_table[pid].euid != 0)
190020     {
190021         errset (EPERM);
190022         return (-1);
190023     }
190024     //
190025     //
190026     //
190027     if (m > 32 || m < 0)
190028     {
190029         errset (EINVAL);
190030         return (-1);
190031     }
190032     //
190033     // Calculate the destination network with the mask.
190034     //
190035     network = ntohl (dest) & ip_mask (m);
190036     //
190037     // Check if there is already. If there is: remove
190038     // it.
190039     //
190040     for (r = 0; r < ROUTE_MAX_ROUTES; r++)
190041     {
190042         if (network == route_table[r].network
190043             && m == route_table[r].m)
190044         {
190045             //
190046             // Remove.
190047             //
190048             memset (&route_table[m], 0xFF,
190049                 sizeof (route_table[m]));
190050             return (0);
190051         }
190052     }
190053     //
190054     // Not found.
190055     //
190056     errset (EINVAL);
190057     return (-1);
190058 }

```

### 94.8.33 kernel/lib\_s/s\_sbrk.c

« Si veda la sezione 87.5.

```

191001 #include <errno.h>
191002 #include <kernel/proc.h>
191003 #include <kernel/lib_k.h>
191004 #include <kernel/lib_s.h>
191005 //-----
191006 void *
191007 s_sbrk (pid_t pid, intptr_t increment)
191008 {
191009     size_t previous_size;
191010     size_t new_size;
191011     int status;
191012     //
191013     // Get current data segment full size.
191014     //
191015     if (proc_table[pid].domain_data == 0)
191016     {
191017         previous_size = (proc_table[pid].domain_text
191018             + proc_table[pid].extra_data);
191019     }
191020     else
191021     {
191022         previous_size = (proc_table[pid].domain_data
191023             + proc_table[pid].extra_data);
191024     }
191025     //
191026     // Check increment.

```

```

191027 //
191028 if ((increment + proc_table[pid].extra_data) < 0)
191029 {
191030     //
191031     // Cannot reduce too much. Just correct it.
191032     //
191033     increment = -proc_table[pid].extra_data;
191034 }
191035 //
191036 // Calculate the new size.
191037 //
191038 new_size = previous_size + increment;
191039 //
191040 // Call 's_brk()' to do the work. The new size value
191041 // is the
191042 // same of the new requested pointer address.
191043 //
191044 status = s_brk (pid, (void *) new_size);
191045 //
191046 if (status < 0)
191047 {
191048     errset (errno);
191049     return ((void *) -1);
191050 }
191051 //
191052 // Ok: return previous final address.
191053 //
191054 return ((void *) previous_size);
191055 }

```

### 94.8.34 kernel/lib\_s/s\_send.c

« Si veda la sezione 87.45.

```

192001 #include <kernel/proc.h>
192002 #include <kernel/net.h>
192003 #include <kernel/net/route.h>
192004 #include <kernel/net/udp.h>
192005 #include <errno.h>
192006 #include <arpa/inet.h>
192007 #include <fcntl.h>
192008 #include <sys/os32.h>
192009 //-----
192010 #define DEBUG 0
192011 //-----
192012 ssize_t
192013 s_send (pid_t pid, int sfdn, const void *buffer,
192014     size_t size, int flags)
192015 {
192016     fd_t *sfd;
192017     int status;
192018     //
192019     // Get file descriptor and verify that it is a
192020     // socket.
192021     //
192022     sfd = fd_reference (pid, &sfdn);
192023     if (sfd == NULL || sfd->file == NULL)
192024     {
192025         errset (EBADF); // Bad file descriptor.
192026         return ((ssize_t) - 1);
192027     }
192028     if (sfd->file->sock == NULL)
192029     {
192030         errset (ENOTSOCK); // Not a socket.
192031         return ((ssize_t) - 1);
192032     }
192033     if (sfd->file->sock->unreach_port)
192034     {
192035         errset (ECONNREFUSED); // Connection refused.
192036         return ((ssize_t) - 1);
192037     }
192038     if (sfd->file->sock->unreach_prot)
192039     {
192040         errset (ENOPROTOOPT); // Protocol not
192041         // available.
192042         return ((ssize_t) - 1);
192043     }
192044     if (sfd->file->sock->unreach_host)
192045     {
192046         errset (EHOSTUNREACH); // Host unreachable.
192047         return ((ssize_t) - 1);
192048     }
192049     if (sfd->file->sock->unreach_net)
192050     {
192051         errset (ENETUNREACH); // Net unreachable.
192052         return ((ssize_t) - 1);
192053     }

```

```

1920054 //
1920055 // Verify to have a valid buffer pointer.
1920056 //
1920057 if (buffer == NULL)
1920058 {
1920059     errset (EINVAL);
1920060     return ((ssize_t) - 1);
1920061 }
1920062 //
1920063 //
1920064 //
1920065 if (sfd->file->sock->family == AF_INET)
1920066 {
1920067     //
1920068     // INET
1920069     //
1920070     // AF_INET requires at least the remote address.
1920071     //
1920072     if (sfd->file->sock->raddr == 0)
1920073     {
1920074         errset (EDESTADDRREQ);
1920075         return ((ssize_t) - 1);
1920076     }
1920077     //
1920078     if (sfd->file->sock->type == SOCK_RAW)
1920079     {
1920080         //
1920081         // RAW
1920082         //
1920083         if (sfd->file->sock->protocol == IPPROTO_ICMP)
1920084         {
1920085             //
1920086             // ICMP
1920087             //
1920088             status = ip_tx (sfd->file->sock->laddr,
1920089                          sfd->file->sock->raddr,
1920090                          sfd->file->sock->protocol,
1920091                          buffer, size);
1920092             if (status)
1920093             {
1920094                 errset (errno);
1920095                 return ((ssize_t) - 1);
1920096             }
1920097             else
1920098             {
1920099                 return ((ssize_t) size);
1920100             }
1920101         }
1920102         else
1920103         {
1920104             //
1920105             // Unsupported protocol.
1920106             //
1920107             errset (EPROTONOSUPPORT);
1920108             return ((ssize_t) - 1);
1920109         }
1920110     }
1920111     else if (sfd->file->sock->type == SOCK_DGRAM)
1920112     {
1920113         //
1920114         // DGRAM
1920115         //
1920116         if (sfd->file->sock->protocol == IPPROTO_UDP)
1920117         {
1920118             //
1920119             // UDP
1920120             //
1920121             status = udp_tx (sfd->file->sock->lport,
1920122                          sfd->file->sock->rport,
1920123                          sfd->file->sock->laddr,
1920124                          sfd->file->sock->raddr,
1920125                          buffer, size);
1920126             if (status)
1920127             {
1920128                 errset (errno);
1920129                 return ((ssize_t) - 1);
1920130             }
1920131             else
1920132             {
1920133                 return ((ssize_t) size);
1920134             }
1920135         }
1920136         else
1920137         {
1920138             //
1920139             // Unsupported protocol.
1920140             //

```

```

1920141     errset (EPROTONOSUPPORT);
1920142     return ((ssize_t) - 1);
1920143 }
1920144 }
1920145 else if (sfd->file->sock->type == SOCK_STREAM)
1920146 {
1920147     //
1920148     // STREAM
1920149     //
1920150     if (sfd->file->sock->protocol == IPPROTO_TCP)
1920151     {
1920152         //
1920153         // TCP
1920154         //
1920155         // See if the send side of the stream
1920156         // was closed.
1920157         //
1920158         if (sfd->file->sock->tcp.send_closed
1920159             || sfd->file->sock->tcp.conn == TCP_CLOSE)
1920160         {
1920161             //
1920162             // End of file.
1920163             //
1920164             if (DEBUG)
1920165             {
1920166                 k_printf ("end of socket write\n");
1920167             }
1920168             s_kill ((pid_t) 0, pid, SIGPIPE);
1920169             errset (EPIPE);
1920170             return ((ssize_t) - 1);
1920171         }
1920172         //
1920173         // Put data to the send buffer, if it is
1920174         // possible.
1920175         //
1920176         if (sfd->file->sock->tcp.can_write)
1920177         {
1920178             size =
1920179                 min (size,
1920180                     (TCP_MSS -
1920181                      sizeof (struct tcphdr)));
1920182             memcpy (sfd->file->sock->tcp.send_data,
1920183                   buffer, size);
1920184             sfd->file->sock->tcp.send_size = size;
1920185             sfd->file->sock->tcp.can_write = 0;
1920186             sfd->file->sock->tcp.can_send = 1;
1920187             //
1920188             sfd->file->sock->tcp.lsq[++sfd->
1920189                          file->sock->tcp.
1920190                          lsqi] =
1920191                 sfd->file->sock->tcp.lsq_ack;
1920192             sfd->file->sock->tcp.send_flags =
1920193                 TCP_FLAG_PSH | TCP_FLAG_ACK;
1920194             tcp_tx_sock (sfd->file->sock);
1920195             //
1920196             return ((ssize_t) size);
1920197         }
1920198     }
1920199     else
1920200     {
1920201         //
1920202         // At the moment, nothing can be
1920203         // written.
1920204         //
1920205         if (sfd->fl_flags & O_NONBLOCK)
1920206         {
1920207             //
1920208             // Cannot block.
1920209             //
1920210             errset (EAGAIN);
1920211             return ((ssize_t) - 1);
1920212         }
1920213         else
1920214         {
1920215             //
1920216             // Go to sleep and return zero.
1920217             //
1920218             proc_table[pid].status =
1920219                 PROC_SLEEPING;
1920220             proc_table[pid].ret = 0;
1920221             proc_table[pid].wakeup_events
1920222                 = WAKEUP_EVENT_SOCKET_WRITE;
1920223             proc_table[pid].wakeup_sock =
1920224                 sfd->file->sock;
1920225             if (DEBUG)
1920226             {
1920227                 k_printf

```

```

1920228         "sleep waiting to write "
1920229         "to a socket.\n",
1920230         __FILE__, pid);
1920231     }
1920232     //
1920233     // Retry.
1920234     //
1920235     errset (EAGAIN);
1920236     return ((ssize_t) - 1);
1920237 }
1920238 }
1920239 }
1920240 else
1920241 {
1920242     //
1920243     // Unsupported protocol.
1920244     //
1920245     errset (EPROTONOSUPPORT);
1920246     return ((ssize_t) - 1);
1920247 }
1920248 }
1920249 else
1920250 {
1920251     //
1920252     // Unsupported type.
1920253     //
1920254     errset (EPROTONOSUPPORT);
1920255     return ((ssize_t) - 1);
1920256 }
1920257 }
1920258 else
1920259 {
1920260     //
1920261     // Unsupported family.
1920262     //
1920263     errset (EAFNOSUPPORT);
1920264     return ((ssize_t) - 1);
1920265 }
1920266 }

```

## 94.8.35 kernel/lib\_s/s\_setegid.c

« Si veda la sezione 87.48.

```

1930001 #include <kernel/proc.h>
1930002 #include <kernel/lib_s.h>
1930003 #include <errno.h>
1930004 //-----
1930005 int
1930006 s_setegid (pid_t pid, gid_t egid)
1930007 {
1930008     if ((proc_table[pid].euid == 0)
1930009         || (proc_table[pid].egid == 0))
1930010     {
1930011         proc_table[pid].egid = egid;
1930012         return (0);
1930013     }
1930014     else if (egid == proc_table[pid].egid)
1930015     {
1930016         return (0);
1930017     }
1930018     else if (egid == proc_table[pid].gid
1930019             || egid == proc_table[pid].sgid)
1930020     {
1930021         proc_table[pid].egid = egid;
1930022         return (0);
1930023     }
1930024     else
1930025     {
1930026         errset (EPERM);
1930027         return (-1);
1930028     }
1930029 }

```

## 94.8.36 kernel/lib\_s/s\_seteuid.c

« Si veda la sezione 87.51.

```

1940001 #include <kernel/proc.h>
1940002 #include <kernel/lib_s.h>
1940003 #include <errno.h>
1940004 //-----
1940005 int
1940006 s_seteuid (pid_t pid, uid_t euid)
1940007 {
1940008     if (proc_table[pid].euid == 0)
1940009     {

```

```

1940010     proc_table[pid].euid = euid;
1940011     return (0);
1940012 }
1940013 else if (euid == proc_table[pid].euid)
1940014 {
1940015     return (0);
1940016 }
1940017 else if (euid == proc_table[pid].uid
1940018         || euid == proc_table[pid].suid)
1940019 {
1940020     proc_table[pid].euid = euid;
1940021     return (0);
1940022 }
1940023 else
1940024 {
1940025     errset (EPERM);
1940026     return (-1);
1940027 }
1940028 }

```

## 94.8.37 kernel/lib\_s/s\_setgid.c

« Si veda la sezione 87.48.

```

1950001 #include <kernel/proc.h>
1950002 #include <kernel/lib_s.h>
1950003 #include <errno.h>
1950004 //-----
1950005 int
1950006 s_setgid (pid_t pid, gid_t gid)
1950007 {
1950008     if ((proc_table[pid].euid == 0)
1950009         || (proc_table[pid].egid == 0))
1950010     {
1950011         proc_table[pid].gid = gid;
1950012         proc_table[pid].egid = gid;
1950013         proc_table[pid].sgid = gid;
1950014         return (0);
1950015     }
1950016     else if (gid == proc_table[pid].egid)
1950017     {
1950018         return (0);
1950019     }
1950020     else if (gid == proc_table[pid].gid
1950021             || gid == proc_table[pid].sgid)
1950022     {
1950023         proc_table[pid].egid = gid;
1950024         return (0);
1950025     }
1950026     else
1950027     {
1950028         errset (EPERM);
1950029         return (-1);
1950030     }
1950031 }

```

## 94.8.38 kernel/lib\_s/s\_setjmp.c

« Si veda la sezione 87.49.

```

1960001 #include <kernel/lib_s.h>
1960002 #include <kernel/proc.h>
1960003 #include <errno.h>
1960004 #include <setjmp.h>
1960005 //-----
1960006 extern uint32_t proc_stack_pointer;
1960007 //-----
1960008 int
1960009 s_setjmp (pid_t pid, jmp_buf env)
1960010 {
1960011     jmp_stack_t *sp;
1960012     jmp_env_t *jmpenv;
1960013     //
1960014     // Find where is the process stack in memory, from
1960015     // the kernel point
1960016     // of view. Please notice that the current stack at
1960017     // 'proc_stack_pointer' will be saved from the
1960018     // scheduler inside
1960019     // the process table, and current stack saved inside
1960020     // the process
1960021     // table is not up to date.
1960022     //
1960023     sp = ptr (pid, (void *) proc_stack_pointer);
1960024     //
1960025     // Translate the pointer 'env', to the kernel point
1960026     // of view.
1960027     //

```

```

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

```

#### 94.8.39 kernel/lib\_s/s\_setuid.c

Si veda la sezione 87.51.

```

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

```

#### 94.8.40 kernel/lib\_s/s\_signal.c

Si veda la sezione 87.52.

```

1980001 #include <kernel/lib_s.h>
1980002 #include <kernel/proc.h>
1980003 #include <errno.h>
1980004 -----
1980005 sighandler_t
1980006 s_signal (pid_t pid, int sig, sighandler_t handler,
1980007          uintptr_t wrapper)
1980008 {
1980009     unsigned long int flag = 1L << (sig - 1);
1980010     sighandler_t previous;
1980011     //
1980012     if (sig <= 0)
1980013     {

```

```

1980014         errset (EINVAL);
1980015         return (SIG_ERR);
1980016     }
1980017     //
1980018     if (proc_table[pid].sig_ignore & flag)
1980019     {
1980020         previous = SIG_IGN;
1980021     }
1980022     else if (proc_table[pid].sig_handler[sig] !=
1980023             (uintptr_t) NULL)
1980024     {
1980025         previous =
1980026             (sighandler_t) proc_table[pid].sig_handler[sig];
1980027     }
1980028     else
1980029     {
1980030         previous = SIG_DFL;
1980031     }
1980032     //
1980033     if (handler == SIG_DFL)
1980034     {
1980035         //
1980036         // Enable signal.
1980037         //
1980038         proc_table[pid].sig_ignore &= ~flag;
1980039         //
1980040         return (previous);
1980041     }
1980042     else if (handler == SIG_IGN)
1980043     {
1980044         //
1980045         // Disable signal.
1980046         //
1980047         proc_table[pid].sig_ignore |= flag;
1980048         //
1980049         return (previous);
1980050     }
1980051     else
1980052     {
1980053         //
1980054         // Enable signal, store the handler address and
1980055         // the
1980056         // handler-wrapper.
1980057         //
1980058         proc_table[pid].sig_ignore &= ~flag;
1980059         proc_table[pid].sig_handler[sig] =
1980060             (uintptr_t) handler;
1980061         proc_table[pid].sig_handler_wrapper = wrapper;
1980062         //
1980063         return (previous);
1980064     }
1980065 }

```

#### 94.8.41 kernel/lib\_s/s\_socket.c

Si veda la sezione 87.54.

```

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

```

```

1990031 // Ok.
1990032 //
1990033 ;
1990034 }
1990035 else
1990036 {
1990037     errset (EPROTONOSUPPORT);
1990038     return (-1);
1990039 }
1990040 //
1990041 // Check supported protocol type.
1990042 //
1990043 if (protocol == IPPROTO_ICMP
1990044     || protocol == IPPROTO_UDP || protocol == IPPROTO_TCP)
1990045 {
1990046     //
1990047     // Ok.
1990048     //
1990049     ;
1990050 }
1990051 else
1990052 {
1990053     errset (EPROTONOSUPPORT);
1990054     return (-1);
1990055 }
1990056 //
1990057 // If it is a raw socket, must be a privileged
1990058 // process.
1990059 //
1990060 if (type == SOCK_RAW && proc_table[pid].euid != 0)
1990061 {
1990062     errset (EACCES);
1990063     return (-1);
1990064 }
1990065 //
1990066 // Find a free slot inside the sock_table[].
1990067 //
1990068 sock = sock_reference (-1);
1990069 if (sock == NULL)
1990070 {
1990071     errset (ENFILE);
1990072     return (-1);
1990073 }
1990074 //
1990075 // Find a free slot inside the file table.
1990076 //
1990077 file = file_reference (-1);
1990078 if (file == NULL)
1990079 {
1990080     errset (ENFILE); // Too many files open in
1990081     // system.
1990082     return (-1);
1990083 }
1990084 //
1990085 // Find a free slot inside the file descriptor
1990086 // table.
1990087 // Variable 'sfdn' will be modified by the call to
1990088 // 'fd_reference()'.
1990089 //
1990090 sfdn = -1;
1990091 fd = fd_reference (pid, &sfdn);
1990092 if (fd == NULL)
1990093 {
1990094     errset (EMFILE); // Too many open files.
1990095     return (-1);
1990096 }
1990097 //
1990098 // socket, system file and file descriptor ready;
1990099 // reset and put data
1990100 // inside them. Please notice that the
1990101 // tcp.listen_queue[] array is
1990102 // reset with all 0xFF, because zero is a valid file
1990103 // descriptor
1990104 // number.
1990105 //
1990106 memset (sock, 0, sizeof (sock_t));
1990107 sock->active = 1;
1990108 sock->family = family;
1990109 sock->type = type;
1990110 sock->protocol = protocol;
1990111 sock->lport = 0;
1990112 sock->laddr = 0;
1990113 sock->rport = 0;
1990114 sock->raddr = 0;
1990115 memset (sock->read.clock, 0x00,
1990116         sizeof (sock->read.clock));
1990117 memset (sock->tcp.listen_queue, 0xFF,

```

```

1990118         sizeof (sock->tcp.listen_queue));
1990119 //
1990120 file->references = 1;
1990121 file->oflags = O_RDWR;
1990122 file->inode = NULL;
1990123 file->sock = sock;
1990124 file->offset = 0;
1990125 //
1990126 fd->fl_flags = (O_RDWR | O_APPEND);
1990127 fd->fd_flags = 0;
1990128 fd->file = file;
1990129 //
1990130 // Return the file descriptor.
1990131 //
1990132 return (sfdn);
1990133 }

```

## 94.8.42 kernel/lib\_s/s\_stat.c

Si veda la sezione 87.55.

```

2000001 #include <kernel/fs.h>
2000002 #include <errno.h>
2000003 #include <kernel/proc.h>
2000004 #include <kernel/lib_s.h>
2000005 //-----
2000006 int
2000007 s_stat (pid_t pid, const char *path, struct stat *buffer)
2000008 {
2000009     proc_t *ps;
2000010     inode_t *inode;
2000011 //
2000012 // Check path.
2000013 //
2000014 if (path == NULL || strlen (path) == 0)
2000015 {
2000016     errset (EINVAL);
2000017     return (-1);
2000018 }
2000019 //
2000020 // Get process.
2000021 //
2000022 ps = proc_reference (pid);
2000023 //
2000024 // Try to load the file inode.
2000025 //
2000026 inode = path_inode (pid, path);
2000027 if (inode == NULL)
2000028 {
2000029     //
2000030     // Cannot access the file: it does not exists or
2000031     // permissions are
2000032     // not sufficient. Variable 'errno' is set by
2000033     // function
2000034     // 'path_inode()'.
2000035     //
2000036     errset (errno);
2000037     return (-1);
2000038 }
2000039 //
2000040 // Inode loaded: update the buffer.
2000041 //
2000042 buffer->st_dev = inode->sb->device;
2000043 buffer->st_ino = inode->ino;
2000044 buffer->st_mode = inode->mode;
2000045 buffer->st_nlink = inode->links;
2000046 buffer->st_uid = inode->uid;
2000047 buffer->st_gid = inode->gid;
2000048 if (S_ISBLK (buffer->st_mode)
2000049     || S_ISCHR (buffer->st_mode))
2000050 {
2000051     buffer->st_rdev = inode->direct[0];
2000052 }
2000053 else
2000054 {
2000055     buffer->st_rdev = 0;
2000056 }
2000057 buffer->st_size = inode->size;
2000058 buffer->st_atime = inode->time; // All times
2000059 // are the
2000060 // same for
2000061 buffer->st_mtime = inode->time; // Minix 1
2000062 // file
2000063 // system.
2000064 buffer->st_ctime = inode->time; //
2000065 buffer->st_blksize = inode->sb->blksize;
2000066 buffer->st_blocks = inode->blkcnt;

```

```

200067 //
200068 // If the inode is a device special file, the
200069 // 'st_rdev' value is
200070 // taken from the first direct zone (as of Minix 1
200071 // organization).
200072 //
200073 if (S_ISBLK (inode->mode) || S_ISCHR (inode->mode))
200074 {
200075     buffer->st_rdev = inode->direct[0];
200076 }
200077 else
200078 {
200079     buffer->st_rdev = 0;
200080 }
200081 //
200082 // Release the inode and return.
200083 //
200084 inode_put (inode);
200085 //
200086 // Return.
200087 //
200088 return (0);
200089 }

```

#### 94.8.43 kernel/lib\_s/s\_stime.c

Si veda la sezione 87.59.

```

200001 #include <kernel/lib_s.h>
200002 #include <kernel/proc.h>
200003 #include <stddef.h>
200004 #include <errno.h>
200005 //-----
200006 extern clock_t _clock_time; // uint64_t
200007 //-----
200008 int
200009 s_stime (pid_t pid, time_t * timer)
200010 {
200011     if (proc_table[pid].euid != 0)
200012     {
200013         errset (EPERM);
200014         return (-1);
200015     }
200016     //
200017     _clock_time = *timer * CLOCKS_PER_SEC;
200018     return (0);
200019 }

```

#### 94.8.44 kernel/lib\_s/s\_tcsetattr.c

Si veda la sezione 87.58.

```

200001 #include <kernel/fs.h>
200002 #include <errno.h>
200003 #include <kernel/proc.h>
200004 #include <kernel/lib_s.h>
200005 #include <termios.h>
200006 #include <sys/types.h>
200007 //-----
200008 int
200009 s_tcsetattr (pid_t pid, int fdn, struct termios *termios_p)
200010 {
200011     file_t *file;
200012     inode_t *inode;
200013     dev_t device;
200014     int t; // 'tty_table[]' subscript.
200015     int c; // 'c_cc[]' subscript.
200016     //
200017     file = proc_table[pid].fd[fdn].file;
200018     //
200019     if (file == NULL)
200020     {
200021         errset (EBADF);
200022         return (-1);
200023     }
200024     //
200025     inode = proc_table[pid].fd[fdn].file->inode;
200026     //
200027     if (inode == NULL)
200028     {
200029         errset (EBADF);
200030         return (-1);
200031     }
200032     //
200033     if (!S_ISCHR (inode->mode))
200034     {
200035         errset (ENOTTY);

```

```

202006     return (-1);
202007 }
202008 //
202009 device = inode->direct[0];
202010 //
202011 if (major (device) != DEV_CONSOLE_MAJOR)
202012 {
202013     errset (ENOTTY);
202014     return (-1);
202015 }
202016 //
202017 t = minor (device);
202018 if (t >= TTYS_TOTAL)
202019 {
202020     errset (ENOTTY);
202021     return (-1);
202022 }
202023 //
202024 // Ok: copy data.
202025 //
202026 termios_p->c_iflag = tty_table[t].attr.c_iflag;
202027 termios_p->c_oflag = tty_table[t].attr.c_oflag;
202028 termios_p->c_cflag = tty_table[t].attr.c_cflag;
202029 termios_p->c_lflag = tty_table[t].attr.c_lflag;
202030 for (c = 0; c < NCCS; c++)
202031 {
202032     termios_p->c_cc[c] = tty_table[t].attr.c_cc[c];
202033 }
202034 //
202035 // Ok.
202036 //
202037 return (0);
202038 }

```

#### 94.8.45 kernel/lib\_s/s\_tcsetattr.c

Si veda la sezione 87.58.

```

200001 #include <kernel/fs.h>
200002 #include <errno.h>
200003 #include <kernel/proc.h>
200004 #include <kernel/lib_s.h>
200005 #include <termios.h>
200006 #include <sys/types.h>
200007 //-----
200008 // The following are masks of the implemented
200009 // attributes.
200010 //
200011 #define MASK_C_IFLAG (BRKINT|ICRN|IGNBRK|IGNCR)
200012 #define MASK_C_OFLAG 0
200013 #define MASK_C_CFLAG 0
200014 #define MASK_C_LFLAG \
200015     (ECHO|ECHOE|ECHOK|ECHONL|ICANON|ISIG)
200016 //-----
200017 int
200018 s_tcsetattr (pid_t pid, int fdn, int action,
200019             struct termios *termios_p)
200020 {
200021     file_t *file;
200022     inode_t *inode;
200023     dev_t device;
200024     int t; // 'tty_table[]' subscript.
200025     int c; // 'c_cc[]' subscript.
200026     //
200027     file = proc_table[pid].fd[fdn].file;
200028     //
200029     if (file == NULL)
200030     {
200031         errset (EBADF);
200032         return (-1);
200033     }
200034     //
200035     inode = proc_table[pid].fd[fdn].file->inode;
200036     //
200037     if (inode == NULL)
200038     {
200039         errset (EBADF);
200040         return (-1);
200041     }
200042     //
200043     if (!S_ISCHR (inode->mode))
200044     {
200045         errset (ENOTTY);
200046         return (-1);
200047     }
200048     //
200049     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

&lt;

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

&lt;

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\_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                 (strcmp (dir->name, name_unlink, NAME_MAX) == 0))

```

```

2000143 {
2000144     //
2000145     // Found the corresponding item: unlink
2000146     // the inode.
2000147     //
2000148     dir->ino = 0;
2000149     //
2000150     // Update the directory inside the file
2000151     // system.
2000152     //
2000153     size_written =
2000154     inode_file_write (inode_directory,
2000155                       start, buffer, size_read);
2000156     if (size_written != size_read)
2000157     {
2000158         //
2000159         // Write problem: just tell.
2000160         //
2000161         k_printf
2000162         ("kernel alert: directory "
2000163          "write error!\n");
2000164     }
2000165     //
2000166     // Update directory inode and put inode.
2000167     // If the unlinked
2000168     // inode was a directory, the parent
2000169     // directory inode
2000170     // must reduce the file system link
2000171     // count.
2000172     //
2000173     if (S_ISDIR (inode_unlink->mode))
2000174     {
2000175         inode_directory->links--;
2000176     }
2000177     inode_directory->time = s_time (pid, NULL);
2000178     inode_directory->changed = 1;
2000179     inode_put (inode_directory);
2000180     //
2000181     // Reduce link inside unlinked inode and
2000182     // put inode.
2000183     //
2000184     inode_unlink->links--;
2000185     inode_unlink->changed = 1;
2000186     inode_unlink->time = s_time (pid, NULL);
2000187     inode_put (inode_unlink);
2000188     //
2000189     // Just return, as the work is done.
2000190     //
2000191     return (0);
2000192 }
2000193 }
2000194 }
2000195 //
2000196 // At this point, it was not possible to unlink the
2000197 // file.
2000198 //
2000199 inode_put (inode_unlink);
2000200 inode_put (inode_directory);
2000201 errset (EUNKNOWN); // Unknown error.
2000202 return (-1);
2000203 }

```

## 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].wakeupt_events =
2070046             WAKEUP_EVENT_SIGNAL;
2070047             proc_table[parent].wakeupt_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               (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     }
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	628
94.9.2	kernel/main/crt0.s	628
94.9.3	kernel/main/kmain.c	629

94.9.4	kernel/main/menu.c	634
94.9.5	kernel/main/run.c	634
94.9.6	kernel/main/stack.s	635

## 94.9.1 kernel/main/build.h

« Si veda la sezione 93.13.

```
210001 #define BUILD_DATE "201108311936"
```

## 94.9.2 kernel/main/crt0.s

« Si veda la sezione 84.2.2.

```
210001 .extern kmain
210002 .extern _k_stack_top
210003 .extern _k_stack_bottom
210004 .global kstartup
210005 #####
210006 #
210007 # The kernel must be compiled as ELF, so that the
210008 # bootloader (GRUB or SYSLINUX) can recognize it.
210009 #
210010 #-----
210011 .section .text
210012 #-----
210013 #
210014 # At the beginning there is the code, but inside the
210015 # code there is also the multiboot header.
210016 # This is why we start with a jump.
210017 #
210018 kstartup:
210019     jmp start
210020 #
210021 # Here is the multiboot header, that must be placed
210022 # near the beginning of the image-file, but aligned at
210023 # a multiple of four bytes.
210024 #
210025 .align 4
210026 multiboot_header:
210027     .int 0x1BADB002          # magic
210028     .int 0x00000007         # flags
210029     .int -(0x1BADB002 + 0x00000007) # checksum
210030 #
210031 # Here starts really the code.
210032 #
210033 start:
210034 #
210035 # Set ESP at the stack bottom.
210036 #
210037 movl $_k_stack_bottom, %esp
210038 #
210039 # Reset flags inside EFLAGS, but must use the stack
210040 # to make it.
210041 #
210042 pushl $0
210043 popf
210044 #
210045 # Call function 'kmain()'. It is not the usual
210046 # 'main()' because we need to pass some data, but
210047 # it is not compatible with the
210048 # standard 'main()' prototype.
210049 #
210050 # void kmain (uint32_t magic, multiboot_t *info);
210051 #
210052 pushl %ebx # Pointer to the structure containing
210053           # data from the boot system.
210054 pushl %eax # Boot system signature.
210055 #
210056 call kmain # Do the call.
210057 #
210058 # Halt procedure.
210059 #
210060 halt:
210061 hlt # If the function 'kmain()' return, this
210062     # instruction will halt the CPU.
210063 jmp halt # If the CPU will resume working, the
210064         # HLT instruction will be repeated.
210065 #-----
210066 .align 4
210067 .section .data
210068 #-----
210069 .align 4
210070 .section .bss
210071 #-----
```

## 94.9.3 kernel/main/kmain.c

« Si veda la sezione 93.13.

```
212001 #include <kernel/main.h>
212002 #include <kernel/main/build.h>
212003 #include <kernel/lib_k.h>
212004 #include <kernel/driver/tty.h>
212005 #include <kernel/memory.h>
212006 #include <stdlib.h>
212007 #include <stdint.h>
212008 #include <kernel/driver/screen.h>
212009 #include <kernel/proc.h>
212010 #include <kernel/lib_s.h>
212011 #include <kernel/fs.h>
212012 #include <unistd.h>
212013 #include <stdint.h>
212014 #include <kernel/driver/kbd.h>
212015 #include <kernel/driver/ata.h>
212016 #include <kernel/dev.h>
212017 #include <kernel/blk.h>
212018 #include <fcntl.h>
212019 #include <string.h>
212020 #include <limits.h>
212021 #include <kernel/driver/pci.h>
212022 #include <kernel/net/icmp.h>
212023 #include <kernel/net/arp.h>
212024 #include <kernel/net/route.h>
212025 #include <kernel/net/tcp.h>
212026 #include <kernel/driver/nic/ne2k.h>
212027 #include <errno.h>
212028 //-----
212029 static char command[MAX_CANON];
212030 //-----
212031 void
212032 kmain (uint32_t magic, multiboot_t * mboot_data)
212033 {
212034     int exit;
212035     pid_t pid;
212036     int counter;
212037     char *exec_argv[2];
212038     int status;
212039     ssize_t count;
212040     //
212041     // 0xAC15FEFE == 172.21.254.254
212042     //
212043     h_addr_t ip_to_be_found = 0xAC15FEFE;
212044     //
212045     // Reset video and select the initial console.
212046     //
212047     tty_init ();
212048     //
212049     // Verify 'multiboot' data.
212050     //
212051     if (magic == 0x2BADB002)
212052     {
212053         //
212054         // Save multiboot data.
212055         //
212056         mboot_save (mboot_data);
212057         //
212058         // Show compilation date and time, plus upper
212059         // memory limit.
212060         //
212061         k_printf ("os32 build %s ram %i Kibyte\n",
212062                 BUILD_DATE, (int) multiboot.mem_upper);
212063         //
212064         // Show also the command line.
212065         //
212066         k_printf ("%s\n", multiboot.cmdline);
212067         //
212068         // Set 'mb_max', for memory allocation.
212069         //
212070         mb_size (multiboot.mem_upper * 1024);
212071         //
212072         // keyboard initialization.
212073         //
212074         kbd_load ();
212075         //
212076         // Block cache initialization.
212077         //
212078         blk_cache_init ();
212079         //
212080         // PCI bus initialization.
212081         //
212082         pci_init ();
212083         //
212084         // File system management initialization.
212085         //
```

```

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 (strcmp (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 (strcmp (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 (strcmp (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 (strcmp (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 (strcmp (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 (strcmp (command, "a", MAX_CANON) == 0)
2120302 {
2120303 run ("/bin/aaa", NULL, NULL);
2120304 }
2120305 else if (strcmp (command, "b", MAX_CANON) == 0)
2120306 {
2120307 run ("/bin/bbb", NULL, NULL);
2120308 }
2120309 else if (strcmp (command, "c", MAX_CANON) == 0)
2120310 {
2120311 run ("/bin/ccc", NULL, NULL);
2120312 }
2120313 else if (strcmp (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 ("%lx", (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 (strcmp (command, "g", MAX_CANON) == 0)
2120343 {
2120344 gdt_print (&gdt_register, 0, 20);
2120345 }
2120346 else if (strcmp (command, "G", MAX_CANON) == 0)

```

```

2120347 {
2120348 gdt_print (&gdt_register, 21, 41);
2120349 }
2120350 else if (strcmp (command, "h", MAX_CANON) == 0)
2120351 {
2120352 menu ();
2120353 }
2120354 else if (strcmp (command, "i", MAX_CANON) == 0)
2120355 {
2120356 idt_print (&idt_register, 0, 20);
2120357 }
2120358 else if (strcmp (command, "I", MAX_CANON) == 0)
2120359 {
2120360 idt_print (&idt_register, 21, 41);
2120361 }
2120362 else if (strcmp (command, "m", MAX_CANON) == 0)
2120363 {
2120364 mb_print ();
2120365 }
2120366 else if (strcmp (command, "n", MAX_CANON) == 0)
2120367 {
2120368 inode_print ();
2120369 }
2120370 else if (strcmp (command, "p", MAX_CANON) == 0)
2120371 {
2120372 proc_print ();
2120373 }
2120374 else if (strcmp (command, "s", MAX_CANON) == 0)
2120375 {
2120376 sb_print ();
2120377 }
2120378 else if (strcmp (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 (strcmp (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 (strcmp (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 (strcmp (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 (strcmp (command, "q", MAX_CANON) == 0)
2120425 {
2120426 k_printf ("System halted!\n");
2120427 return;
2120428 }
2120429 else if (strcmp (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, 0x010000
2150011 #-----
2150012 .align 4
2150013 .section .bss
2150014 #-----
2150015 #
2150016 # At the end is placed the space for the kernel stack,
2150017 # with no initialization.
2150018 #
2150019 _k_stack_top:
2150020 .space STACK_SIZE
2150021 _k_stack_bottom:
2150022 #-----

```

#### 94.10 os32: «kernel/memory.h»

Si veda la sezione 93.14.

```

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

```

94.10.1	kernel/memory/mb_alloc.c	636
94.10.2	kernel/memory/mb_alloc_size.c	637
94.10.3	kernel/memory/mb_clean.c	638
94.10.4	kernel/memory/mb_free.c	638
94.10.5	kernel/memory/mb_print.c	639
94.10.6	kernel/memory/mb_public.c	640
94.10.7	kernel/memory/mb_reduce.c	640
94.10.8	kernel/memory/mb_reference.c	641
94.10.9	kernel/memory/mb_size.c	641



## 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 static int
2170084 mb_block_set1 (int block)
2170085

```

```

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 }
218100 }
218101 //-----
218102 static int
218103 mb_block_status (int block)
218104 {
218105     int i = block / 32;
218106     int j = block % 32;
218107     uint32_t mask = 0x80000000 >> j;
218108     return ((int) (mb_table[i] & mask));
218109 }
218110 }

```

### 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 static int
2200082 mb_block_set0 (int block)
2200083 {
2200084     int i = block / 32;
2200085     int j = block % 32;
2200086     uint32_t mask = 0x80000000 >> j;
2200087     if (mb_table[i] & mask)
2200088     {
2200089         mb_table[i] = mb_table[i] & ~mask;
2200090         return (1);
2200091     }
2200092     else
2200093     {
2200094         return (0); // The block is already set to
2200095                     // 0 inside the map!
2200096     }
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 .....642

94.11.2 kernel/multiboot/mboot\_public.c ..... 643

94.11.3 kernel/multiboot/mboot\_save.c ..... 643

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.

<

```

2280001 #include <kernel/multiboot.h>
2280002 #include <string.h>
2280003 #include <kernel/lib_k.h>
2280004 //-----
2280005 void
2280006 mboot_save (multiboot_t * mboot_data)
2280007 {
2280008     multiboot.flags = mboot_data->flags;
2280009     //
2280010     if ((mboot_data->flags & 1) > 0)
2280011     {
2280012         multiboot.mem_lower = mboot_data->mem_lower;
2280013         multiboot.mem_upper = mboot_data->mem_upper;
2280014     }
2280015     if ((mboot_data->flags & 2) > 0)
2280016     {
2280017         multiboot.boot_device = mboot_data->boot_device;
2280018     }
2280019     if ((mboot_data->flags & 4) > 0)
2280020     {
2280021         strncpy (multiboot.cmdline, mboot_data->cmdline,
2280022                 MBOOT_CMDLINE_MAX);
2280023     }
2280024     else
2280025     {
2280026         memset (multiboot.cmdline, 0, MBOOT_CMDLINE_MAX);
2280027     }
2280028 }

```

## 94.12 os32: «kernel/net.h»

« Si veda la sezione 93.17.

```

230001 #ifndef _KERNEL_NET_H
230002 #define _KERNEL_NET_H 1
230003 //-----
230004 //
230005 // WARNING: please remember to limit the max packets
230006 // size to be transmitted, to a value that ensure no
230007 // fragmentation at IPv4 level.
230008 //
230009 //-----
230010 #include <stdint.h>
230011 #include <sys/types.h>
230012 #include <netinet/in.h>
230013 #include <lib/sys/os32.h>
230014 #include <netinet/ip.h>
230015 #include <arpa/inet.h>
230016 //-----
230017 #define NET_PROT_IP 0x0800
230018 #define NET_PROT_ARP 0x0806
230019 #define NET_PROT_RARP 0x8035
230020 //
230021 #define NET_ETHERNET_ADDRESS_LENGTH 6
230022 #define NET_IP_ADDRESS_LENGTH 4
230023 //
230024 #define NET_DEV_NULL 0x0000
230025 #define NET_DEV_LOOP 0x0100
230026 #define NET_DEV_LOOPBACK 0x0101
230027 #define NET_DEV_ETH 0x0200
230028 #define NET_DEV_ETH_NE2K 0x0201
230029 //
230030 #define NET_MAX_DEVICES 4
230031 #define NET_MAX_BUFFERS 64
230032 //
230033 #define NET_ETHERNET_MIN_PACKET_SIZE 46
230034 #define NET_ETHERNET_MAX_PACKET_SIZE 1500
230035 #define NET_ETHERNET_HEADER_SIZE 14
230036 #define NET_ETHERNET_MAX_FRAME_SIZE \
230037 (NET_ETHERNET_MAX_PACKET_SIZE \
230038 +NET_ETHERNET_HEADER_SIZE)
230039 //
230040 #define NET_IP_MIN_HEADER_SIZE 20
230041 #define NET_IP_MIN_PACKET_SIZE 576
230042 #define NET_IP_MAX_PACKET_SIZE \
230043 NET_ETHERNET_MAX_PACKET_SIZE // [1]
230044 #define NET_IP_MAX_DATA_SIZE \
230045 (NET_IP_MAX_PACKET_SIZE \
230046 -NET_IP_MIN_HEADER_SIZE)
230047 //
230048 // [1] The IP max packet size should be 65535, but as
230049 // os32 does not accept fragmented packets, the
230050 // maximum value depends on the Ethernet packet
230051 // size. But this limitation might change, so
230052 // there is also the value NET_MTU, that depends
230053 // from the real physical network limitations.
230054 //
230055 #define NET_MTU NET_ETHERNET_MAX_PACKET_SIZE
230056 //-----
230057 //
230058 // IP unfragmented packet, but it is not known if there
230059 // are IP options, so it is not possible to define the
230060 // space after the minimal header.
230061 //
230062 typedef union
230063 {
230064     uint8_t octet[NET_IP_MAX_PACKET_SIZE];
230065     struct iphdr header;
230066     __attribute__((packed)) net_ip_packet_t;
230067 //
230068 // Ethernet header.
230069 //
230070 typedef struct
230071 {
230072     uint8_t dst[6];
230073     uint8_t src[6];
230074     uint16_t type;
230075     } __attribute__((packed)) net_ethernet_header_t;
230076 //
230077 // Ethernet frame.
230078 //
230079 //-----
230080 // | .
230081 // |-----|
230082 // | .octet[]
230083 // |-----|
230084 // | .header | .packet

```

```

230085 // |-----|
230086 // | ... | .packet.octet[]
230087 // |-----|
230088 // | | .packet.header | | | | | | | | | | |
230089 // |-----|
230090 //
230091 typedef union
230092 {
230093 //
230094     uint8_t octet[NET_ETHERNET_MAX_FRAME_SIZE];
230095 //
230096     struct
230097     {
230098         net_ethernet_header_t header;
230099         union
230100         {
230101             //
230102             uint8_t octet[NET_IP_MAX_PACKET_SIZE];
230103             //
230104             struct iphdr header;
230105             //
230106             } __attribute__((packed)) packet;
230107         } __attribute__((packed));
230108 //
230109     } __attribute__((packed)) net_ethernet_frame_t;
230110 //-----
230111 //
230112 // Ethernet buffer.
230113 //
230114 typedef struct
230115 {
230116     clock_t clock;
230117     size_t size;
230118     net_ethernet_frame_t frame;
230119     } __attribute__((packed)) net_buffer_eth_t;
230120 //
230121 // Loopback buffer.
230122 //
230123 typedef struct
230124 {
230125     clock_t clock;
230126     size_t size;
230127     union
230128     {
230129         uint8_t octet[NET_MTU];
230130         struct iphdr header;
230131         } packet;
230132     } __attribute__((packed)) net_buffer_lo_t; // [2]
230133 //
230134 // [2] The structure net_ip_packet_t is not used here,
230135 // because it refers to unfragmented packets, where
230136 // here the max size might be less.
230137 //
230138 //-----
230139 //
230140 // Network interfaces table structure
230141 //
230142 typedef struct
230143 {
230144     unsigned int type;
230145     h_addr_t ip; // IPv4 address in host byte order.
230146     uint8_t m; // Short netmask.
230147     union
230148     {
230149         //
230150         // Ethernet type data:
230151         //
230152         struct
230153         {
230154             uint8_t mac[6];
230155             uintptr_t base_io;
230156             unsigned char irq;
230157             net_buffer_eth_t buffer[NET_MAX_BUFFERS];
230158         } ethernet;
230159         //
230160         // Loopback type data:
230161         //
230162         struct
230163         {
230164             net_buffer_lo_t buffer[NET_MAX_BUFFERS];
230165         } loopback;
230166     };
230167     } net_t;
230168 //
230169 // [2] The structure net_ip_packet_t is not used here,
230170 // because it refers to unfragmented packets, where
230171 // here the max size might be less.

```

```

2300172 //
2300173 extern net_t net_table[NET_MAX_DEVICES];
2300174 //-----
2300175 int net_rx (void);
2300176 int net_tcp (void);
2300177 void net_init (void);
2300178 int net_index (h_addr_t ip);
2300179 int net_index_eth (h_addr_t ip, uint8_t mac[6],
2300180                  uintptr_t io);
2300181
2300182 net_buffer_eth_t *net_buffer_eth (int n);
2300183 net_buffer_lo_t *net_buffer_lo (int n);
2300184 void net_print (void);
2300185
2300186 //void net_eth_init          (int start);
2300187 int net_eth_tx (int dev, void *buffer, size_t size);
2300188 int net_eth_ip_tx (h_addr_t src, h_addr_t dst,
2300189                  const void *packet, size_t size);
2300190 //-----
2300191 #endif

```

94.12.1	kernel/net/arp.h	647
94.12.2	kernel/net/arp/arp_clean.c	648
94.12.3	kernel/net/arp/arp_index.c	648
94.12.4	kernel/net/arp/arp_init.c	649
94.12.5	kernel/net/arp/arp_print.c	649
94.12.6	kernel/net/arp/arp_public.c	649
94.12.7	kernel/net/arp/arp_reference.c	649
94.12.8	kernel/net/arp/arp_request.c	650
94.12.9	kernel/net/arp/arp_rx.c	651
94.12.10	kernel/net/icmp.h	653
94.12.11	kernel/net/icmp/icmp_rx.c	653
94.12.12	kernel/net/icmp/icmp_tx.c	655
94.12.13	kernel/net/icmp/icmp_tx_echo.c	656
94.12.14	kernel/net/icmp/icmp_tx_unreachable.c	656
94.12.15	kernel/net/ip.h	657
94.12.16	kernel/net/ip/ip_checksum.c	658
94.12.17	kernel/net/ip/ip_header.c	659
94.12.18	kernel/net/ip/ip_mask.c	659
94.12.19	kernel/net/ip/ip_public.c	660
94.12.20	kernel/net/ip/ip_reference.c	660
94.12.21	kernel/net/ip/ip_rx.c	660
94.12.22	kernel/net/ip/ip_tx.c	663
94.12.23	kernel/net/net_buffer_eth.c	665
94.12.24	kernel/net/net_buffer_lo.c	665
94.12.25	kernel/net/net_eth_ip_tx.c	666
94.12.26	kernel/net/net_eth_tx.c	668
94.12.27	kernel/net/net_index.c	668
94.12.28	kernel/net/net_index_eth.c	668
94.12.29	kernel/net/net_init.c	669
94.12.30	kernel/net/net_print.c	672
94.12.31	kernel/net/net_public.c	672
94.12.32	kernel/net/net_rx.c	673
94.12.33	kernel/net/route.h	674
94.12.34	kernel/net/route/route_init.c	674
94.12.35	kernel/net/route/route_print.c	675
94.12.36	kernel/net/route/route_public.c	675
94.12.37	kernel/net/route/route_remote_to_local.c	676
94.12.38	kernel/net/route/route_remote_to_router.c	676

94.12.39	kernel/net/route/route_sort.c	677
94.12.40	kernel/net/tcp.h	679
94.12.41	kernel/net/tcp/tcp.c	679
94.12.42	kernel/net/tcp/tcp_close.c	690
94.12.43	kernel/net/tcp/tcp_connect.c	691
94.12.44	kernel/net/tcp/tcp_rx_ack.c	692
94.12.45	kernel/net/tcp/tcp_rx_data.c	693
94.12.46	kernel/net/tcp/tcp_show.c	695
94.12.47	kernel/net/tcp/tcp_status.c	695
94.12.48	kernel/net/tcp/tcp_test.c	696
94.12.49	kernel/net/tcp/tcp_tx_ack.c	697
94.12.50	kernel/net/tcp/tcp_tx_raw.c	698
94.12.51	kernel/net/tcp/tcp_tx_rst.c	699
94.12.52	kernel/net/tcp/tcp_tx_sock.c	700
94.12.53	kernel/net/udp.h	702
94.12.54	kernel/net/udp/udp_tx.c	702

## 94.12.1 kernel/net/arp.h

Si veda la sezione 93.1.

```

2310001 #ifndef _KERNEL_NET_ARP_H
2310002 #define _KERNEL_NET_ARP_H 1
2310003 //-----
2310004 #include <stdint.h>
2310005 #include <sys/types.h>
2310006 #include <kernel/net/ip.h>
2310007 #include <arpa/inet.h>
2310008 //-----
2310009 #define ARP_HW_ETHERNET 1
2310010 #define ARP_HW_IEEE802 6 // Example, but not
2310011                          // used.
2310012 //-----
2310013 #define ARP_TYPE_REQUEST 1
2310014 #define ARP_TYPE_REPLY 2
2310015 //-----
2310016 typedef struct
2310017 {
2310018     uint16_t hardware_type;
2310019     uint16_t protocol_type;
2310020     uint8_t hardware_address_length;
2310021     uint8_t protocol_address_length;
2310022     uint16_t opcode;
2310023     uint8_t sender_mac[NET_ETHERNET_ADDRESS_LENGTH];
2310024     uint32_t sender_ip; // Network byte order.
2310025     uint8_t target_mac[NET_ETHERNET_ADDRESS_LENGTH];
2310026     uint32_t target_ip; // Network byte order.
2310027     uint8_t filler[18]; // [1]
2310028 } __attribute__((packed)) arp_packet_t;
2310029 //
2310030 // [1] The filler is just big enough to get a minimal
2310031 //      Ethernet frame size.
2310032 //
2310033 //-----
2310034 #define ARP_MAX_ITEMS 64
2310035 #define ARP_MAX_TIME 300 // Seconds.
2310036 //
2310037 typedef struct
2310038 {
2310039     time_t time;
2310040     uint8_t mac[NET_ETHERNET_ADDRESS_LENGTH];
2310041     h_addr_t ip; // Host byte order.
2310042 } arp_t;
2310043 //
2310044 extern arp_t arp_table[ARP_MAX_ITEMS];
2310045 //-----
2310046 void arp_clean (void);
2310047 int arp_index (unsigned char mac[6], h_addr_t ip);
2310048 void arp_init (void);
2310049 void arp_print (void);
2310050 arp_t *arp_reference (void);
2310051 void arp_request (h_addr_t ip);
2310052 int arp_rx (int n, int f);
2310053 //-----
2310054 #endif

```

## 94.12.2 kernel/net/arp/arp\_clean.c

« Si veda la sezione 93.1.

```

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

```

## 94.12.3 kernel/net/arp/arp\_index.c

« Si veda la sezione 93.1.

```

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

```

## 94.12.4 kernel/net/arp/arp\_init.c

« Si veda la sezione 93.1.

```

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

```

## 94.12.5 kernel/net/arp/arp\_print.c

« Si veda la sezione 93.1.

```

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

```

## 94.12.6 kernel/net/arp/arp\_public.c

« Si veda la sezione 93.1.

```

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

```

## 94.12.7 kernel/net/arp/arp\_reference.c

« Si veda la sezione 93.1.

```

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

```

```

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

```

## 94.12.8 kernel/net/arp/arp\_request.c

« Si veda la sezione 93.1.

```

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

```

## 94.12.9 kernel/net/arp/arp\_rx.c

« Si veda la sezione 93.1.

```

2390001 #include <kernel/net.h>
2390002 #include <kernel/net/arp.h>
2390003 #include <kernel/net/ip.h>
2390004 #include <sys/os32.h>
2390005 #include <kernel/lib_k.h>
2390006 #include <errno.h>
2390007 #include <arpa/inet.h>
2390008 //-----
2390009 #define DEBUG 0
2390010 //-----
2390011 int
2390012 arp_rx (int n, int f)
2390013 {
2390014     net_ethernet_frame_t *frame =
2390015     &net_table[n].ethernet.buffer[f].frame;
2390016     arp_packet_t *arp = (arp_packet_t *) &frame->packet;
2390017     int i;
2390018     int a; // ARP table index.
2390019     arp_t *arp_table_new_item;
2390020     //
2390021     net_ethernet_frame_t ans_frame;
2390022     arp_packet_t *ans_arp =
2390023     (arp_packet_t *) &ans_frame.packet;
2390024     //
2390025     //
2390026     //
2390027     if (n >= NET_MAX_DEVICES || n < 0)
2390028     {
2390029         errset (EINVAL); // Invalid argument.
2390030         return (-1);
2390031     }
2390032     //
2390033     if (!(net_table[n].type & NET_DEV_ETH))
2390034     {
2390035         errset (EINVAL); // Invalid argument.
2390036         return (-1);
2390037     }
2390038     //
2390039     if (ntohs (frame->header.type) != NET_PROT_ARP)
2390040     {
2390041         errset (EINVAL); // Invalid argument.
2390042         return (-1);
2390043     }
2390044     //
2390045     //
2390046     //
2390047     if (ntohs (arp->opcode) == ARP_TYPE_REQUEST)
2390048     {
2390049         //
2390050         // This is an ARP request: we try to answer 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 answer.
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             //

```



```

239086 // Send it.
239087 //
239088 net_eth_tx (n, &ans_frame,
239089           sizeof (arp_packet_t) + 14);
239090 }
239091 //
239092 // Done.
239093 //
239094 return (0);
239095 }
239096 //
239097 //
239098 //
239099 if (ntohs (arp->opcode) == ARP_TYPE_REPLY)
239100 {
239101 //
239102 // We should save it inside the ARP table.
239103 //
239104 for (a = 0; a < ARP_MAX_ITEMS; a++)
239105 {
239106 //
239107 // Check if we already have the same item.
239108 //
239109 if (memcmp
239110     (arp->sender_mac, arp_table[a].mac,
239111      (size_t) NET_ETHERNET_ADDRESS_LENGTH) == 0)
239112 {
239113     if (arp_table[a].ip == ntohl (arp->sender_ip))
239114     {
239115 //
239116 // Found: update the time.
239117 //
239118 arp_table[a].time = k_time (NULL);
239119 //
239120 // Done.
239121 //
239122 return (0);
239123 }
239124 else
239125 {
239126 //
239127 // There is already the Ethernet
239128 // address,
239129 // but the IP is different.
239130 //
239131 if (DEBUG)
239132 {
239133     k_printf
239134     ("%s:%i: Ethernet address "
239135      "%x02:%02:%x02:%x02:%02:%x02 "
239136      "has more than one IP\n",
239137      __FILE__, __LINE__,
239138      arp_table[a].mac[0],
239139      arp_table[a].mac[1],
239140      arp_table[a].mac[2],
239141      arp_table[a].mac[3],
239142      arp_table[a].mac[4],
239143      arp_table[a].mac[5]);
239144 //
239145 // End of scan.
239146 //
239147 break;
239148 }
239149 }
239150 }
239151 //
239152 //
239153 // If we are here, the MAC-IP couple is new: get
239154 // a new ARP item.
239155 //
239156 arp_table_new_item = arp_reference ();
239157 //
239158 memcpy (arp_table_new_item->mac, arp->sender_mac,
239159         (size_t) NET_ETHERNET_ADDRESS_LENGTH);
239160 arp_table_new_item->ip = ntohl (arp->sender_ip);
239161 arp_table_new_item->time = k_time (NULL);
239162 //
239163 // Done.
239164 //
239165 return (0);
239166 }
239167 //
239168 // If we are here, we don't know the ARP type.
239169 //
239170 if (DEBUG)
239171 {
239172     k_printf ("%s:%i: unknown ARP type: %i\n",

```

```

239073           (int) ntohs (arp->opcode));
239074 }
239075 //
239076 // Done.
239077 //
239078 return (0);
239079 }

```

#### 94.12.10 kernel/net/icmp.h

Si veda la sezione 93.8.

```

240001 #ifndef _KERNEL_NET_ICMP_H
240002 #define _KERNEL_NET_ICMP_H 1
240003 //-----
240004 #include <stdint.h>
240005 #include <sys/types.h>
240006 #include <kernel/net/ip.h>
240007 #include <netinet/icmp.h>
240008 //-----
240009 #define ICMP_HEADER_SIZE 8
240010 #define ICMP_MAX_PACKET_SIZE NET_IP_MAX_DATA_SIZE
240011 #define ICMP_MAX_DATA_SIZE \
240012     ICMP_MAX_PACKET_SIZE-ICMP_HEADER_SIZE
240013 //-----
240014 //
240015 // ICMP packet, for transmission.
240016 //
240017 typedef struct
240018 {
240019     struct icmp_hdr header;
240020     uint8_t data[ICMP_MAX_DATA_SIZE];
240021 } __attribute__ ((packed)) icmp_packet_t;
240022 //-----
240023 //
240024 int icmp_rx (int i);
240025 //
240026 int icmp_tx (h_addr_t src, h_addr_t dst,
240027             int type, int code, icmp_packet_t * icmp,
240028             size_t size);
240029 //
240030 int icmp_tx_echo (h_addr_t src, h_addr_t dst,
240031                 int type, int code,
240032                 int identifier, int sequence,
240033                 uint8_t * data, size_t size);
240034 //
240035 int icmp_tx_unreachable (h_addr_t src, h_addr_t dst,
240036                         int type, int code,
240037                         uint8_t * data, size_t size);
240038 //
240039 //-----
240040 #endif

```

#### 94.12.11 kernel/net/icmp/icmp\_rx.c

Si veda la sezione 93.8.

```

241001 #include <sys/os32.h>
241002 #include <kernel/lib_k.h>
241003 #include <arpa/inet.h>
241004 #include <kernel/net.h>
241005 #include <kernel/net/ip.h>
241006 #include <kernel/net/icmp.h>
241007 #include <netinet/icmp.h>
241008 #include <netinet/udp.h>
241009 #include <kernel/lib_k.h>
241010 #include <errno.h>
241011 //-----
241012 #define DEBUG 0
241013 //-----
241014 int
241015 icmp_rx (int i)
241016 {
241017     icmp_packet_t *icmp;
241018     size_t size;
241019     struct ip_hdr *ip;
241020     struct udphdr *ports;
241021     h_addr_t dest = 0;
241022     h_port_t port = 0;
241023     int s; // Socket table index.
241024 //
241025 //
241026 //
241027 if ((i >= IP_MAX_PACKETS) || i < 0)
241028 {
241029     errset (EINVAL);
241030     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 icmp_hdr);
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 ip_hdr *)
2410096             ((uint8_t *) icmp) + sizeof (struct icmp_hdr));
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     else if (icmp->header.type == ICMP_ECHOREPLY)
2410161     {
2410162         //
2410163         // ECHO REPLY
2410164         //
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

&lt;&lt;

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 icmp_hdr) + size)));
2430038 }

```

#### 94.12.14 kernel/net/icmp/icmp\_tx\_unreachable.c

&lt;&lt;

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 icmp_hdr) + size)));
2440055 }

```

#### 94.12.15 kernel/net/ip.h

&gt;&gt;

Si veda la sezione 93.9.

```

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

```

```

2450049 // to have see a certain packet and that the kernel
2450050 // does not have to try to service it again. For
2450051 // example, when an ECHO REQUEST packet arrive, the
2450052 // kernel answers with an ECHO REPLY, but does not
2450053 // remove the packet that might be read from a true
2450054 // RAW socket. So, the kernel writes inside
2450055 // kernel_serviced the same value found inside
2450056 // clock, so that it know that it was already read.
2450057 //
2450058 //
2450059 // External IP table data.
2450060 //
2450061 extern ip_t ip_table[IP_MAX_PACKETS];
2450062 //-----
2450063 uint16_t ip_checksum (uint16_t * data1, size_t size1,
2450064                      uint16_t * data2, size_t size2);
2450065
2450066 int ip_rx (int n, int f);
2450067 ip_t *ip_reference (void);
2450068 ssize_t ip_header (h_addr_t src, h_addr_t dst,
2450069                  uint16_t id, uint8_t ttl,
2450070                  uint8_t protocol, void *buffer,
2450071                  size_t length);
2450072
2450073 int ip_tx (h_addr_t src, h_addr_t dst, int protocol,
2450074           const void *buffer, size_t size);
2450075 h_addr_t ip_mask (int m);
2450076 //-----
2450077 #endif

```

#### 94.12.16 kernel/net/ip/ip\_checksum.c

« Si veda la sezione 93.9.

```

2460001 #include <stdint.h>
2460002 #include <arpa/inet.h>
2460003 #include <kernel/net/ip.h>
2460004 //-----
2460005 uint16_t
2460006 ip_checksum (uint16_t * data1, size_t size1,
2460007             uint16_t * data2, size_t size2)
2460008 {
2460009     int i;
2460010     uint32_t sum;
2460011     uint16_t carry;
2460012     uint16_t checksum;
2460013     uint16_t last;
2460014     uint8_t *octet;
2460015     //
2460016     // 2's complement sum.
2460017     //
2460018     sum = 0;
2460019     //
2460020     if (data1 != NULL)
2460021     {
2460022         for (i = 0; i < (size1 / 2); i++)
2460023         {
2460024             sum += ntohs (data1[i]);
2460025         }
2460026         //
2460027         if (size1 % 2)
2460028         {
2460029             //
2460030             // The size is odd, and the last octet must
2460031             // be accounted too.
2460032             //
2460033             octet = (uint8_t *) data1;
2460034             last = octet[size1 - 1];
2460035             last = last << 8;
2460036             sum += last;
2460037         }
2460038     }
2460039     if (data2 != NULL)
2460040     {
2460041         for (i = 0; i < (size2 / 2); i++)
2460042         {
2460043             sum += ntohs (data2[i]);
2460044         }
2460045         //
2460046         if (size2 % 2)
2460047         {
2460048             //
2460049             // The size is odd, and the last octet must
2460050             // be accounted too.
2460051             //
2460052             octet = (uint8_t *) data2;
2460053             last = octet[size2 - 1];
2460054             last = last << 8;
2460055             sum += last;

```

```

2460056     }
2460057 }
2460058 //
2460059 // Extract the carries and make the checksum.
2460060 //
2460061 carry = sum >> 16;
2460062 checksum = sum & 0x0000FFFF;
2460063 checksum += carry;
2460064 //
2460065 // End of job.
2460066 //
2460067 return (checksum);
2460068 }

```

#### 94.12.17 kernel/net/ip/ip\_header.c

« Si veda la sezione 93.9.

```

2470001 #include <kernel/net.h>
2470002 #include <kernel/net/ip.h>
2470003 #include <sys/os32.h>
2470004 #include <kernel/lib_k.h>
2470005 #include <errno.h>
2470006 #include <arpa/inet.h>
2470007 #include <netinet/ip.h>
2470008 //-----
2470009 #define DEBUG 0
2470010 //-----
2470011 ssize_t
2470012 ip_header (h_addr_t src, h_addr_t dst, uint16_t id,
2470013           uint8_t ttl, uint8_t protocol, void *buffer,
2470014           size_t length)
2470015 {
2470016     struct iphdr header;
2470017     uint16_t checksum;
2470018     //
2470019     // Check size.
2470020     //
2470021     if (length < sizeof (struct iphdr))
2470022     {
2470023         errset (EINVAL);
2470024         return ((ssize_t) - 1);
2470025     }
2470026     //
2470027     // Prepare the header.
2470028     //
2470029     header.version = IP_VERSION;
2470030     header.ihl = sizeof (struct iphdr) / 4;
2470031     header.tos = 0x00; // Routine, normal.
2470032     header.tot_len = htons (length);
2470033     header.id = htons (id);
2470034     header.frag_off = htons (0x4000); // Do not
2470035     // fragment!
2470036     header.ttl = ttl;
2470037     header.protocol = protocol;
2470038     header.check = 0;
2470039     header.saddr = htonl (src);
2470040     header.daddr = htonl (dst);
2470041     //
2470042     // Fix the length.
2470043     //
2470044     length = sizeof (struct iphdr);
2470045     //
2470046     // Now set the header checksum.
2470047     //
2470048     checksum = ~(ip_checksum ((void *) &header, length,
2470049                             NULL, (size_t) 0));
2470050     header.check = htons (checksum);
2470051     //
2470052     // Copy the header to the buffer
2470053     //
2470054     memcpy (buffer, &header, length);
2470055     //
2470056     // Return size written.
2470057     //
2470058     return (length);
2470059 }

```

#### 94.12.18 kernel/net/ip/ip\_mask.c

« Si veda la sezione 93.9.

```

2480001 #include <kernel/net.h>
2480002 #include <kernel/net/route.h>
2480003 #include <kernel/net/arp.h>
2480004 //-----
2480005 h_addr_t

```

```

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

```

#### 94.12.19 kernel/net/ip/ip\_public.c

« Si veda la sezione 93.9.

```

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

```

#### 94.12.20 kernel/net/ip/ip\_reference.c

« Si veda la sezione 93.9.

```

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

```

#### 94.12.21 kernel/net/ip/ip\_rx.c

« Si veda la sezione 93.9.

```

2510001 #include <kernel/net.h>
2510002 #include <kernel/net/arp.h>
2510003 #include <kernel/net/icmp.h>
2510004 #include <kernel/net/ip.h>
2510005 #include <sys/os32.h>
2510006 #include <kernel/lib_k.h>
2510007 #include <errno.h>
2510008 #include <arpa/inet.h>
2510009 #include <netinet/udp.h>
2510010 //-----
2510011 #define DEBUG 0
2510012 //-----
2510013 int
2510014 ip_rx (int n, int f)
2510015 {
2510016     struct iphdr *header;
2510017     net_ip_packet_t *packet;
2510018     uint16_t checksum;
2510019     size_t size_header;

```

```

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

```

```

2510107 else
2510108 {
2510109 //
2510110 // Find a place inside the IP table.
2510111 //
2510112 ip_table_item = ip_reference ();
2510113 //
2510114 // Copy the packet inside the ip_table[] item,
2510115 // then update
2510116 // the link to the data start inside the packet
2510117 // and the
2510118 // clock_t timestamp.
2510119 //
2510120 memcpy (ip_table_item->packet.octet, packet,
2510121         ntohs (header->tot_len));
2510122 ip_table_item->pdu4 = ip_table_item->packet.octet;
2510123 ip_table_item->pdu4 += (header->ihl * 4);
2510124 ip_table_item->clock = k_clock ();
2510125 }
2510126 //
2510127 // Check for destination unreachable, scanning the
2510128 // interface table,
2510129 // to see if there is such address here.
2510130 //
2510131 for (j = 0; j < NET_MAX_DEVICES; j++)
2510132 {
2510133     if (net_table[j].ip == ntohl (header->daddr))
2510134     {
2510135         //
2510136         // Found a valid local address.
2510137         //
2510138         break;
2510139     }
2510140 }
2510141 if (j >= NET_MAX_DEVICES)
2510142 {
2510143     //
2510144     // Local address not found: host unreachable,
2510145     // but the packet
2510146     // is taken anyway.
2510147     //
2510148     icmp_tx_unreachable (ntohl (header->daddr),
2510149                          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_served !=
2510204             ip_table[i].clock
2510205             && ip_table[i].packet.header.protocol ==
2510206                 IPPROTO_ICMP)
2510207         {
2510208             icmp_rx (i);
2510209         }
2510210     }
2510211     else
2510212     {
2510213         //
2510214         // At the moment, no other IP protocol
2510215         // managed internally.
2510216         ip_table[i].kernel_served =
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 //-----

```

```

254008 net_buffer_lo_t *
254009 net_buffer_lo (int n)
254010 {
254011     int b;           // Buffer index.
254012     int ref = -1;   // Reference index.
254013     clock_t clock = k_clock (); // Reference clock
254014     // value.
254015     //
254016     // Check NET table index.
254017     //
254018     if ((n > NET_MAX_DEVICES) || (n < 0))
254019     {
254020         errset (EINVAL);
254021         return (NULL);
254022     }
254023     //
254024     if (!(net_table[n].type & NET_DEV_LOOP))
254025     {
254026         errset (EINVAL);
254027         return (NULL);
254028     }
254029     //
254030     // Loopback found.
254031     //
254032     for (b = 0; b < NET_MAX_BUFFERS; b++)
254033     {
254034         if (net_table[n].loopback.buffer[b].clock == 0)
254035         {
254036             //
254037             // Enough.
254038             //
254039             return &net_table[n].loopback.buffer[b];
254040         }
254041         else if (net_table[n].loopback.buffer[b].clock <
254042                 clock)
254043         {
254044             clock = net_table[n].loopback.buffer[b].clock;
254045             ref = b;
254046         }
254047     }
254048     //
254049     // Return the selected frame structure.
254050     //
254051     return &net_table[n].loopback.buffer[ref];
254052     //
254053     // Device not found!
254054     //
254055     errset (ENODEV);
254056     return (NULL);
254057 }

```

#### 94.12.25 kernel/net/net\_eth\_ip\_tx.c

« Si veda la sezione 93.17.

```

255001 #include <sys/os32.h>
255002 #include <kernel/net/route.h>
255003 #include <kernel/net/ip.h>
255004 #include <kernel/net/arp.h>
255005 #include <kernel/driver/nic/ne2k.h>
255006 #include <kernel/driver/pci.h>
255007 #include <kernel/ibm_i386.h>
255008 #include <errno.h>
255009 //-----
255010 int
255011 net_eth_ip_tx (h_addr_t src, h_addr_t dst,
255012               const void *packet, size_t size)
255013 {
255014     net_ethernet_frame_t frame;
255015     int n;           // NET table index.
255016     int a;           // ARP table index.
255017     int i;
255018     h_addr_t router;
255019     //
255020     // Check for PDU size.
255021     //
255022     if (size > NET_ETHERNET_MAX_PACKET_SIZE)
255023     {
255024         errset (E_PDU_TOO_BIG);
255025         return (-1);
255026     }
255027     //
255028     // Find the sender interface address.
255029     //
255030     n = net_index_eth (src, NULL, (uintptr_t) 0);
255031     if (n < 0)
255032     {

```

```

255033         errset (errno); // ENODEV.
255034         return (-1);
255035     }
255036     //
255037     // Copy the Ethernet source address into the
255038     // Ethernet frame
255039     // header.
255040     //
255041     memcpy (frame.header.src, net_table[n].ethernet.mac,
255042            NET_ETHERNET_ADDRESS_LENGTH);
255043     //
255044     // Find if we need a router.
255045     //
255046     router = route_remote_to_router (dst);
255047     //
255048     if (router != 0 && router != ((h_addr_t) - 1))
255049     {
255050         //
255051         // We need to find the router destination MAC
255052         // address.
255053         //
255054         a = arp_index (NULL, router);
255055         if (a < 0)
255056         {
255057             //
255058             // There is not the item inside the ARP
255059             // table. Send a request
255060             // and return.
255061             //
255062             arp_request (router);
255063             errset (E_ARP_MISSING);
255064             return (-1);
255065         }
255066     }
255067     else
255068     {
255069         //
255070         // The destination is inside the local network.
255071         // Find the destination Ethernet address.
255072         //
255073         a = arp_index (NULL, dst);
255074         if (a < 0)
255075         {
255076             //
255077             // There is not the item inside the ARP
255078             // table. Send a request
255079             // and return.
255080             //
255081             arp_request (dst);
255082             errset (E_ARP_MISSING);
255083             return (-1);
255084         }
255085     }
255086     //
255087     // Copy the Ethernet destination address into the
255088     // Ethernet frame
255089     // header: might be the real destination interface,
255090     // or the
255091     // router.
255092     //
255093     memcpy (frame.header.dst, arp_table[a].mac,
255094            NET_ETHERNET_ADDRESS_LENGTH);
255095     //
255096     // Set the frame type.
255097     //
255098     frame.header.type = htons (NET_PROT_IP);
255099     //
255100     // Copy the IP packet.
255101     //
255102     memcpy (&frame.packet, packet, size);
255103     //
255104     // Fill if the size is too little.
255105     //
255106     for (i = size; i < NET_ETHERNET_MIN_PACKET_SIZE; i++)
255107     {
255108         frame.packet.octet[i] = 0;
255109     }
255110     //
255111     size = max (size, NET_ETHERNET_MIN_PACKET_SIZE);
255112     //
255113     // Now, send the Ethernet frame. Index 'n' is the
255114     // network
255115     // device number.
255116     //
255117     return (net_eth_tx
255118            (n, &frame, size + NET_ETHERNET_HEADER_SIZE));
255119 }

```



## 94.12.26 kernel/net/net\_eth\_tx.c

« Si veda la sezione 93.17.

```

258001 #include <sys/os32.h>
258002 #include <kernel/net.h>
258003 #include <kernel/driver/nic/ne2k.h>
258004 #include <kernel/driver/pci.h>
258005 #include <kernel/ibm_i386.h>
258006 #include <errno.h>
258007 //-----
258008 int
258009 net_eth_tx (int n, void *buffer, size_t size)
258010 {
258011     //
258012     if (n >= NET_MAX_DEVICES || n < 0)
258013     {
258014         errset (EINVAL);
258015         return (-1);
258016     }
258017     if (!(net_table[n].type & NET_DEV_ETH))
258018     {
258019         errset (EINVAL);
258020         return (-1);
258021     }
258022     //
258023     if (net_table[n].type == NET_DEV_ETH_NE2K)
258024     {
258025         return (ne2k_tx
258026             (net_table[n].ethernet.base_io, buffer,
258027             size));
258028     }
258029     //
258030     // If we are here, there is not the driver for the
258031     // Ethernet device.
258032     //
258033     errset (ENODEV);
258034     return (-1);
258035 }

```

## 94.12.27 kernel/net/net\_index.c

« Si veda la sezione 93.17.

```

257001 #include <sys/os32.h>
257002 #include <kernel/net.h>
257003 #include <errno.h>
257004 #include <stdint.h>
257005 #include <arpa/inet.h>
257006 //-----
257007 int
257008 net_index (h_addr_t ip)
257009 {
257010     //
257011     int n;
257012     //
257013     // By IPv4 address.
257014     //
257015     if (ip != 0)
257016     {
257017         for (n = 0; n < NET_MAX_DEVICES; n++)
257018         {
257019             if (net_table[n].ip == ip)
257020             {
257021                 return (n);
257022             }
257023         }
257024     }
257025     //
257026     // Not found!
257027     //
257028     errset (ENODEV);
257029     return (-1);
257030 }

```

## 94.12.28 kernel/net/net\_index\_eth.c

« Si veda la sezione 93.17.

```

258001 #include <sys/os32.h>
258002 #include <kernel/net.h>
258003 #include <errno.h>
258004 #include <stdint.h>
258005 #include <arpa/inet.h>
258006 //-----
258007 int
258008 net_index_eth (h_addr_t ip, uint8_t mac[6], uintptr_t io)
258009 {

```

```

258010 //
258011 int n;
258012 //
258013 // If 'ip' is not zero, then find the Ethernet table
258014 // index by that
258015 // value.
258016 //
258017 if (ip != 0)
258018 {
258019     for (n = 0; n < NET_MAX_DEVICES; n++)
258020     {
258021         if (net_table[n].type & NET_DEV_ETH)
258022         {
258023             if (net_table[n].ip == ip)
258024             {
258025                 return (n);
258026             }
258027         }
258028     }
258029 }
258030 //
258031 // By mac address.
258032 //
258033 if (mac != NULL)
258034 {
258035     for (n = 0; n < NET_MAX_DEVICES; n++)
258036     {
258037         if (net_table[n].type & NET_DEV_ETH)
258038         {
258039             if (net_table[n].ethernet.mac[0] ==
258040                 mac[0]
258041                 && net_table[n].ethernet.mac[1] ==
258042                 mac[1]
258043                 && net_table[n].ethernet.mac[2] ==
258044                 mac[2]
258045                 && net_table[n].ethernet.mac[3] ==
258046                 mac[3]
258047                 && net_table[n].ethernet.mac[4] ==
258048                 mac[4]
258049                 && net_table[n].ethernet.mac[5] == mac[5])
258050             {
258051                 return (n);
258052             }
258053         }
258054     }
258055 }
258056 //
258057 // By hardware I/O address.
258058 //
258059 if (io > 0)
258060 {
258061     for (n = 0; n < NET_MAX_DEVICES; n++)
258062     {
258063         if (net_table[n].type & NET_DEV_ETH)
258064         {
258065             if (net_table[n].ethernet.base_io == io)
258066             {
258067                 return (n);
258068             }
258069         }
258070     }
258071 }
258072 //
258073 // Not found!
258074 //
258075 errset (ENODEV);
258076 return (-1);
258077 }

```

## 94.12.29 kernel/net/net\_init.c

« Si veda la sezione 93.17.

```

259001 #include <kernel/net.h>
259002 #include <kernel/net/route.h>
259003 #include <kernel/net/arp.h>
259004 #include <kernel/lib_s.h>
259005 #include <kernel/proc.h>
259006 #include <kernel/multiboot.h>
259007 #include <stdlib.h>
259008 #include <kernel/lib_k.h>
259009 #include <string.h>
259010 #include <errno.h>
259011 #include <kernel/driver/pci.h>
259012 #include <kernel/driver/nic/ne2k.h>
259013 //-----
259014 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_routead (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                     " "
2600028                     net_table[n].ip >> 24 & 0x000000FF,
2600029                     net_table[n].ip >> 16 & 0x000000FF,
2600030                     net_table[n].ip >> 8 & 0x000000FF,
2600031                     net_table[n].ip >> 0 & 0x000000FF,
2600032                     net_table[n].m);
2600033             string[20] = '\0';
2600034             k_printf ("%s", string);
2600035             //
2600036             if (net_table[n].type & NET_DEV_ETH)
2600037             {
2600038                 k_printf
2600039                     ("%02x:%02x:%02x:%02x:%02x:%02x "
2600040                     "0x%04x %i",
2600041                     net_table[n].ethernet.mac[0],
2600042                     net_table[n].ethernet.mac[1],
2600043                     net_table[n].ethernet.mac[2],
2600044                     net_table[n].ethernet.mac[3],
2600045                     net_table[n].ethernet.mac[4],
2600046                     net_table[n].ethernet.mac[5],
2600047                     net_table[n].ethernet.base_io,
2600048                     net_table[n].ethernet.irq);
2600049             }
2600050             k_printf ("\n");
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         }

```

```

2620886 //
2620887 // Loopback
2620888 //
2620889 else if (net_table[n].type & NET_DEV_LOOP)
2620890 {
2620891     for (b = 0; b < NET_MAX_BUFFERS; b++)
2620892     {
2620893         if (net_table[n].loopback.buffer[b].clock > 0)
2620894         {
2620895             ip_rx (n, b);
2620896             //
2620897             // Remove packet from buffer.
2620898             //
2620899             net_table[n].loopback.buffer[b].clock = 0;
2620900             //
2620901             // Increment the packet received
2620902             // counter.
2620903             //
2620904             counter++;
2620905             //
2620906         }
2620907     }
2620908 }
2620909 }
2620910 //
2620911 // Remove ARP items that are too old.
2620912 //
2620913 arp_clean ();
2620914 //
2620915 //
2620916 //
2620917 return (counter);
2620918 }

```

#### 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 static int
2690024 comp (void *a, void *b)
2690025 {
2690026     route_t *route_a = a;
2690027     route_t *route_b = b;
2690028     uint8_t m_a = route_a->m;
2690029     uint8_t m_b = route_b->m;
2690030     //
2690031     if (m_a > m_b)
2690032         return (-1);
2690033     if (m_a < m_b)
2690034         return (1);
2690035     return (0);
2690036 }
2690037 //-----
2690038 static void
2690039 qsort (void *base, size_t nmemb, size_t size,
2690040       int (*compare) (void *, void *))
2690041 {
2690042     if (size <= 1)
2690043     {
2690044         //
2690045         // There is nothing to sort!
2690046         //
2690047         return;
2690048     }
2690049     else
2690050     {
2690051         sort ((char *) base, size, 0, (int) (nmemb - 1),
2690052              compare);
2690053     }
2690054 }
2690055 //-----
2690056 static void
2690057 sort (char *array, size_t size, int a, int z,
2690058      int (*compare) (void *, void *))
2690059 {
2690060     int loc;
2690061     //
2690062     if (z > a)
2690063     {
2690064         loc = part (array, size, a, z, compare);
2690065         if (loc >= 0)

```

```

260069      {
260070          sort (array, size, a, loc - 1, compare);
260071          sort (array, size, loc + 1, z, compare);
260072      }
260073  }
260074 }
260075
260076 //-----
260077 static int
260078 part (char *array, size_t size, int a, int z,
260079      int (*compare) (void *, void *))
260080 {
260081     int i;
260082     int loc;
260083     //
260084     if (z <= a)
260085     {
260086         errset (EUNKNOWN);          // Should never
260087         // happen.
260088         return (-1);
260089     }
260090     //
260091     // Index 'i' after the first element; index 'loc' at
260092     // the last
260093     // position.
260094     //
260095     i = a + 1;
260096     loc = z;
260097     //
260098     // Loop as long as index 'loc' is higher than index
260099     // 'i'.
260100     // When index 'loc' is less or equal to index 'i',
260101     // then, index 'loc' is the right position for the
260102     // first element of the current piece of array.
260103     //
260104     for (;;)
260105     {
260106         //
260107         // Index 'i' goes up...
260108         //
260109         for (; i < loc; i++)
260110             {
260111                 if (compare
260112                     (&array[i * size], &array[a * size]) > 0)
260113                 {
260114                     break;
260115                 }
260116             }
260117         //
260118         // Index 'loc' goes down...
260119         //
260120         for (; loc-- > i)
260121             {
260122                 if (compare
260123                     (&array[loc * size], &array[a * size]) <= 0)
260124                 {
260125                     break;
260126                 }
260127             }
260128         //
260129         // Swap elements related to index 'i' and 'loc'.
260130         //
260131         if (loc <= i)
260132             {
260133                 //
260134                 // The array is completely scanned.
260135                 //
260136                 break;
260137             }
260138         else
260139             {
260140                 memcpy (swap, &array[loc * size], size);
260141                 memcpy (&array[loc * size], &array[i * size],
260142                         size);
260143                 memcpy (&array[i * size], swap, size);
260144             }
260145     }
260146     //
260147     // Swap the first element with the one related to
260148     // the
260149     // index 'loc'.
260150     //
260151     memcpy (swap, &array[loc * size], size);
260152     memcpy (&array[loc * size], &array[a * size], size);
260153     memcpy (&array[a * size], swap, size);
260154     //
260155     // Return the index 'loc'.

```

```

260156 //
260157     return (loc);
260158 }

```

## 94.12.40 kernel/net/tcp.h

Si veda la sezione 93.23.

```

270001 #ifndef _KERNEL_NET_TCP_H
270002 #define _KERNEL_NET_TCP_H 1
270003 //-----
270004 #include <netinet/tcp.h>
270005 #include <kernel/net.h>
270006 //-----
270007 #define TCP_HEADER_SIZE 20
270008 #define TCP_MAX_PACKET_SIZE NET_IP_MAX_DATA_SIZE
270009 #define TCP_MAX_DATA_SIZE \
270010     TCP_MAX_PACKET_SIZE-TCP_HEADER_SIZE
270011 //
270012 #define TCP_MAX_DELAY (CLOCKS_PER_SEC*2)
270013 //-----
270014 //
270015 // TCP packet, for transmission.
270016 //
270017 typedef struct
270018 {
270019     struct tcphdr header;
270020     uint8_t data[TCP_MAX_DATA_SIZE];
270021 } __attribute__((packed)) tcp_packet_t;
270022 //
270023 // TCP pseudo header for checksum calculation.
270024 //
270025 typedef struct
270026 {
270027     in_addr_t saddr;
270028     in_addr_t daddr;
270029     uint8_t zero;
270030     uint8_t protocol;
270031     uint16_t length;
270032 } __attribute__((packed)) tcp_pseudo_header_t;
270033 //-----
270034 #define TCP_FLAG_NULL 0
270035 #define TCP_FLAG_ACK 1
270036 #define TCP_FLAG_PSH 2
270037 #define TCP_FLAG_RST 4
270038 #define TCP_FLAG_SYN 8
270039 #define TCP_FLAG_FIN 16
270040 //-----
270041 #define TCP_TRY_READ 1 // 2^0 Wake up reading
270042 // TCP process.
270043 #define TCP_TRY_WRITE 2 // 2^1 Wake up writing
270044 // TCP process.
270045 //-----
270046 int tcp_tx_raw (h_port_t sport, h_port_t dport,
270047                uint32_t seq, uint32_t ack_seq,
270048                int flags,
270049                h_addr_t saddr, h_addr_t daddr,
270050                const void *buffer, size_t size);
270051 int tcp_tx_sock (void *sock_item);
270052 int tcp_tx_ack (void *sock_item);
270053 int tcp_rx_ack (void *sock_item, void *packet);
270054 int tcp (void);
270055 int tcp_connect (void *sock_item);
270056 int tcp_tx_rst (void *ip_packet);
270057 void tcp_test (void);
270058 void tcp_show (h_addr_t src, h_addr_t dst,
270059                const struct tcphdr *tcphdr);
270060 int tcp_close (void *sock_item);
270061 int tcp_rx_data (void *sock_item, void *packet);
270062 int tcp_status (void *ip_packet);
270063 //-----
270064 #endif

```

## 94.12.41 kernel/net/tcp/tcp.c

Si veda la sezione 93.23.

```

271001 #include <stdlib.h>
271002 #include <string.h>
271003 #include <netinet/ip.h>
271004 #include <netinet/tcp.h>
271005 #include <kernel/net.h>
271006 #include <kernel/net/tcp.h>
271007 #include <kernel/fs.h>
271008 #include <kernel/lib_s.h>
271009 #include <kernel/lib_k.h>
271010 #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 the packet
2710151         // might be a SYN.
2710152         //
2710153         if (sock_table[s].raddr == 0
2710154             && sock_table[s].tcp.conn ==
2710155                 TCP_LISTEN && tcp->syn && !tcp->ack)
2710156         {
2710157             //
2710158             // We hope that it is the first SYN.
2710159             //
2710160             ;
2710161         }
2710162         else
2710163         {
2710164             continue;
2710165         }
2710166     }
2710167     //
2710168     // This TCP packet is new for the socket:
2710169     // save the clock time, so that the
2710170     // same packet is not read again.
2710171     //
2710172     sock_table[s].read.clock[p] = ip_table[p].clock;
2710173     //
2710174     // //////////////////////////////////////
2710175     // TCP PROTOCOL
2710176     // //////////////////////////////////////
2710177     //
2710178     if (DEBUG)
2710179     {
2710180         tcp_show (ntohl (ip->saddr),
2710181                 ntohl (ip->daddr), tcp);
2710182     }
2710183     //
2710184     recv_data = &(uint8_t *) tcp[tcp->doff + 4];

```

```

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

```

```

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

```



```

2710359 // easily.
2710360 //
2710361 sfd =
2710362     fd_reference (sock_table[s].tcp.
2710363                 listen_pid, &sfdn);
2710364 sock = sfd->file->sock;
2710365 //
2710366 // Connect the new socket with the
2710367 // remote node.
2710368 //
2710369 sock->raddr = ntohl (ip->saddr);
2710370 sock->rport = ntohs (tcp->source);
2710371 //
2710372 // The new socket has seen this SYN
2710373 // packet.
2710374 //
2710375 sock->read.clock[p] = ip_table[p].clock;
2710376 //
2710377 // Make the new socket answere 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_rcv = 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 }
2710489 else if (sock_table[s].tcp.conn == TCP_SYN_RECV)
2710490 {
2710491     //
2710492     // We are waiting an ACK for our second
2710493     // SYN.
2710494     //
2710495     if (tcp->ack
2710496         && tcp_rx_ack (&sock_table[s], ip) == 0)
2710497     {
2710498         //
2710499         // ACK ok.
2710500         // The connection is ready.
2710501         //
2710502         sock_table[s].tcp.conn = TCP_ESTABLISHED;
2710503         //
2710504         // Now the process can write and can
2710505         // receive
2710506         // data (can write --but cannot
2710507         // send-- and can
2710508         // receive --but cannot read--).
2710509         //
2710510         sock_table[s].tcp.can_write = 1;
2710511         sock_table[s].tcp.can_send = 0;
2710512         //
2710513         sock_table[s].tcp.can_rcv = 1;
2710514         sock_table[s].tcp.can_read = 0;
2710515         //
2710516         ret |= TCP_TRY_WRITE;
2710517     }
2710518     //
2710519     // No other type of packet is expected
2710520     // here.
2710521     //
2710522 }
2710523 }
2710524 else if (sock_table[s].tcp.conn ==
2710525         TCP_ESTABLISHED)
2710526 {
2710527     //
2710528     // Might be a repeated SYN + ACK,
2710529     // because the other
2710530     // side don't have received our ACK.
2710531     // Just resend.
2710532     //
2710533     if (tcp->syn && tcp->ack)
2710534     {

```

```

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             ntohs (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             ntohs (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     }
2710746     else
2710747     {
2710748         //
2710749         // Next packet.
2710750         //
2710751         continue;
2710752     }
2710753     //
2710754     // Is the FIN in the right sequence?
2710755     //
2710756     if (sock_table[s].tcp.
2710757         rsq[sock_table[s].tcp.rsqi] ==
2710758         ntohl (tcp->seq))
2710759     {
2710760         //
2710761         // Yes, it is: close receiving.
2710762         //
2710763         sock_table[s].tcp.recv_closed = 1;
2710764         //
2710765         // ACK.
2710766         //
2710767         sock_table[s].tcp.
2710768             rsq[++sock_table[s].tcp.rsqi] =
2710769             ntohl (tcp->seq) + 1;
2710770         sock_table[s].tcp.
2710771             lsq[++sock_table[s].tcp.lsqi] =
2710772             sock_table[s].tcp.lsq_ack;
2710773         tcp_tx_ack (&sock_table[s]);
2710774         //
2710775         // Change status.
2710776         //
2710777         if (DEBUG)
2710778             {
2710779                 k_printf
2710780                     ("[%s] TCP_TIME_WAIT\n",
2710781                     __func__);
2710782             }
2710783         sock_table[s].tcp.conn =
2710784             TCP_TIME_WAIT;
2710785     }
2710786     else
2710787     {
2710788         //
2710789         // Just ignore it and jump to
2710790         // the next packet.
2710791         //
2710792         continue;
2710793     }
2710794 }

```

```

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     }
2710811     else
2710812     {
2710813         //
2710814         // Next packet.
2710815         //
2710816         continue;
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     }
2710851     else
2710852     {
2710853         //
2710854         // Next packet.
2710855         //
2710856         continue;
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.

```

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_close (void *sock_item)
2730014 {
2730015     sock_t *sock = sock_item;
2730016     //
2730017     // Is there already a connection?
2730018     //
2730019     if (sock->tcp.conn == 0 || sock->tcp.conn == TCP_CLOSE)
2730020     {
2730021         //
2730022         // Done or never opened.
2730023         //
2730024         return (0);
2730025     }
2730026     else if (sock->tcp.conn == TCP_RESET)
2730027     {
2730028         //
2730029         // Change to closed.
2730030         //
2730031         if (DEBUG)
2730032         {
2730033             k_printf ("%s] TCP_CLOSE\n", __func__);
2730034         }
2730035         sock->tcp.conn = TCP_CLOSE;
2730036         return (0);
2730037     }
2730038     else if (sock->tcp.conn == TCP_TIME_WAIT)
2730039     {
2730040         //
2730041         // Assume that the time is elapsed.
2730042         //
2730043         sock->tcp.conn = TCP_CLOSE;
2730044         if (DEBUG)
2730045         {
2730046             k_printf ("%s] TCP_CLOSE\n", __func__);
2730047         }
2730048         sock->tcp.conn = TCP_CLOSE;
2730049         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     if (i >= 16)
2740039     {
2740040         if (DEBUG)
2740041         {
2740042             k_printf ("ERR loc seq: ");
2740043             tcp_show (ntohl (iphdr->saddr),
2740044                     ntohl (iphdr->daddr), tcphdr);
2740045         }
2740046         int j;
2740047         for (j = 0; j < 16; j++)
2740048         {
2740049             if (sock->tcp.lsq[j] != 0)
2740050             {
2740051                 k_printf ("%3u ", sock->tcp.lsq[j]);
2740052             }
2740053             k_printf ("lsq_ack=%3u ", sock->tcp.lsq_ack);
2740054             k_printf ("\n");
2740055         }
2740056         //
2740057         // The ACK is out of sequence: sorry.
2740058         //
2740059         tcp_tx_rst (iphdr);
2740060     }
2740061     return (-1);
2740062 }
2740063 //
2740064 // Is the TCP sequence what we expected? But notice
2740065 // that, if our
2740066 // remote expected sequence is zero, this is the
2740067 // first time that
2740068 // get it, so it is right.
2740069 //
2740070 if (sock->tcp.rsq[sock->tcp.rsqi] != 0
2740071     && sock->tcp.rsq[sock->tcp.rsqi] !=
2740072     ntohl (tcphdr->seq))
2740073 {
2740074 //
2740075 // If it is a previous sequence, just ignore
2740076 // the packet.
2740077 //
2740078 for (i = 0; i < 16; i++)
2740079 {
2740080     if (sock->tcp.rsq[i] == ntohl (tcphdr->seq))
2740081     {
2740082         break;
2740083     }
2740084 }
2740085 if (i >= 16)
2740086 {
2740087 //
2740088 // The packet is out of sequence: sorry.
2740089 //
2740090 if (DEBUG)
2740091 {
2740092     k_printf ("ERR rem seq: ");
2740093     tcp_show (ntohl (iphdr->saddr),
2740094             ntohl (iphdr->daddr), tcphdr);
2740095 }
2740096 int j;
2740097 for (j = 0; j < 16; j++)
2740098 {
2740099     if (sock->tcp.rsq[j] != 0)
2740100     {
2740101         k_printf ("%3u ", sock->tcp.rsq[j]);
2740102     }
2740103     k_printf ("\n");
2740104 }
2740105     tcp_tx_rst (iphdr);
2740106 }
2740107 return (-1);
2740108 }
2740109 return (0);
2740110 }
2740111 }
2740112 }
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.rsq] =
2750080         ntohl (tcphdr->seq) + recv_size;
2750081     sock->tcp.lsq[++sock->tcp.lsq] = 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);

```

```

2760093 }

```

#### 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.

```

280001 #include <kernel/net.h>
280002 #include <kernel/net/ip.h>
280003 #include <kernel/net/route.h>
280004 #include <kernel/net/tcp.h>
280005 #include <sys/os32.h>
280006 #include <kernel/lib_k.h>
280007 #include <errno.h>
280008 #include <arpa/inet.h>
280009 #include <netinet/tcp.h>
280010 //-----
280011 #define DEBUG 0
280012 //-----
280013 int
280014 tcp_tx_raw (h_port_t sport, h_port_t dport,
280015             uint32_t seq, uint32_t ack_seq, int flags,
280016             h_addr_t saddr, h_addr_t daddr,
280017             const void *buffer, size_t size)
280018 {
280019     tcp_packet_t packet;
280020     tcp_pseudo_header_t pseudo;
280021     uint16_t checksum;
280022     size_t tcp_size = size + sizeof (struct tcphdr);
280023     //
280024     // Verify to have the source address: it is
280025     // necessary here for
280026     // the checksum calculation.
280027     //
280028     if (saddr == 0)
280029     {
280030         //
280031         // Default source address: get the source
280032         // address from the routing
280033         // table, based on the destination.
280034         //
280035         saddr = route_remote_to_local (daddr);
280036         if (saddr == ((h_addr_t) - 1))
280037         {
280038             errset (errno);
280039             return (-1);
280040         }
280041     }
280042     //
280043     // Prepare the TCP packet.
280044     //
280045     memset (&packet.header, 0, sizeof (struct tcphdr));
280046     //
280047     packet.header.source = htons (sport);
280048     packet.header.dest = htons (dport);
280049     packet.header.seq = htonl (seq);
280050     packet.header.ack_seq = htonl (ack_seq);
280051     packet.header.doff = (sizeof (struct tcphdr) / 4);
280052     if (flags & TCP_FLAG_ACK)
280053         packet.header.ack = 1;
280054     if (flags & TCP_FLAG_PSH)
280055         packet.header.psh = 1;
280056     if (flags & TCP_FLAG_RST)
280057         packet.header.rst = 1;
280058     if (flags & TCP_FLAG_SYN)
280059         packet.header.syn = 1;
280060     if (flags & TCP_FLAG_FIN)
280061         packet.header.fin = 1;
280062     if (flags & TCP_FLAG_RST)
280063     {
280064         packet.header.window = htons (0);
280065     }
280066     else
280067     {
280068         //
280069         // Minimal window.
280070         //
280071         packet.header.window = htons (TCP_MSS);
280072     }
280073     packet.header.check = 0;
280074     //
280075     memcpy (packet.data, buffer, size);
280076     //
280077     // Prepare the pseudo header.
280078     //
280079     pseudo.saddr = htonl (saddr);
280080     pseudo.daddr = htonl (daddr);
280081     pseudo.zero = 0;
280082     pseudo.protocol = IPPROTO_TCP;
280083     pseudo.length = htons (tcp_size);
280084     //
280085     // Now set the header checksum.

```

```

280086     //
280087     checksum = ~(ip_checksum ((void *) &packet, tcp_size,
280088                             (void *) &pseudo,
280089                             sizeof (tcp_pseudo_header_t)));
280090     if (checksum == 0)
280091     {
280092         checksum = 0xFFFF;
280093     }
280094     packet.header.check = htons (checksum);
280095     //
280096     // Send to the lower network level.
280097     //
280098     return (ip_tx
280099             (saddr, daddr, (int) IPPROTO_TCP, &packet,
280100             tcp_size));
280101 }

```

## 94.12.51 kernel/net/tcp/tcp\_tx\_rst.c

Si veda la sezione 93.23.

```

281001 #include <kernel/net.h>
281002 #include <kernel/net/ip.h>
281003 #include <kernel/net/route.h>
281004 #include <kernel/net/tcp.h>
281005 #include <sys/os32.h>
281006 #include <kernel/lib_k.h>
281007 #include <kernel/fs.h>
281008 #include <errno.h>
281009 #include <stdlib.h>
281010 #include <arpa/inet.h>
281011 #include <netinet/in.h>
281012 #include <netinet/tcp.h>
281013 //-----
281014 #define DEBUG 0
281015 //-----
281016 int
281017 tcp_tx_rst (void *ip_packet)
281018 {
281019     struct iphdr *iphdr = ip_packet;
281020     struct tcphdr *tcphdr;
281021     uint32_t seq;
281022     uint32_t ack_seq;
281023     tcp_packet_t packet;
281024     tcp_pseudo_header_t pseudo;
281025     uint16_t checksum;
281026     //
281027     if (ip_packet == NULL)
281028     {
281029         errset (EINVAL);
281030         return (-1);
281031     }
281032     //
281033     iphdr = ip_packet;
281034     tcphdr = (struct tcphdr *)
281035             &(((uint8_t *) ip_packet)[iphdr->ihl * 4]);
281036     //
281037     if (iphdr->saddr == 0 || iphdr->daddr == 0)
281038     {
281039         errset (EINVAL);
281040         return (-1);
281041     }
281042     //
281043     if (tcphdr->source == 0 || tcphdr->dest == 0)
281044     {
281045         errset (EINVAL);
281046         return (-1);
281047     }
281048     //
281049     // If the bad TCP packet has a ACK sequence, we
281050     // replay with
281051     // the same sequence (the one that the other side
281052     // expects).
281053     //
281054     if (tcphdr->ack)
281055     {
281056         seq = ntohl (tcphdr->ack_seq);
281057     }
281058     else
281059     {
281060         seq = rand ();
281061     }
281062     //
281063     // Our reset ACK has the same sequence received.
281064     //
281065     ack_seq = ntohl (tcphdr->seq);
281066     //

```



```

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)
2821000     packet.header.syn = 1;
2821001 if (sock->tcp.send_flags & TCP_FLAG_FIN)
2821002     packet.header.fin = 1;
2821003 if (sock->tcp.send_flags & TCP_FLAG_RST)
2821004     {
2821005         packet.header.window = htons (0);
2821006     }
2821007 else
2821008     {
2821009         //
2821010         // Minimal window.
2821011         //
2821012         packet.header.window = htons (TCP_MSS);
2821013     }
2821014 packet.header.check = 0;
2821015 //
2821016 memcpy (packet.data, sock->tcp.send_data, send_size);
2821017 //
2821018 tcp_size = sizeof (struct tcphdr) + send_size;
2821019 //
2821020 // Prepare the pseudo header.
2821021 //
2821022 pseudo.saddr = htonl (sock->laddr);
2821023 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 = 0xFFFFF;
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 tcp_hdr *) &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 = 0xFFFFF;
2840065 }
2840066 packet.header.check = htons (checksum);
2840067 //
2840068 // Send to the lower network level.
2840069 //
2840070 return (ip_tx
2840071 (saddr, daddr, (int) IPPROTO_UDP, &packet,
2840072 udp_size));
2840073 }

```

### 94.13 os32: «kernel/part.h»

« Si veda la sezione 93.18.

```

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

```

```

2850024 #define PART_ACTIVE      0x80 // value for active
2850025 // partition
2850026 #define PART_MAX        4 // number of entries
2850027 // in partition table
2850028 #define PART_TABLE_OFF  0x1BE // offset of part.
2850029 // table in boot
2850030 // sector
2850031
2850032 //
2850033 // Partition types.
2850034 //
2850035 #define PART_TYPE_NONE  0x00 // unused
2850036 // entry
2850037 #define PART_TYPE_NO_PART 0xFF // full device
2850038 #define PART_TYPE_MINIX  0x81 // Minix
2850039 // partition
2850040 // type
2850041 #define PART_TYPE_OLDMINIX 0x80 // Minix 1 old
2850042 // partition
2850043 // type
2850044 #define PART_TYPE_EXT    0x05 // extended
2850045 // partition
2850046
2850047
2850048 #endif

```

## 94.14 os32: «kernel/proc.h»

« Si veda la sezione 93.20.

```

2860001 #ifndef _KERNEL_PROC_H
2860002 #define _KERNEL_PROC_H 1
2860003 //-----
2860004 #include <kernel/ibm_i386.h>
2860005 #include <kernel/dev.h>
2860006 #include <kernel/driver/tty.h>
2860007 #include <sys/types.h>
2860008 #include <sys/stat.h>
2860009 #include <kernel/fs.h>
2860010 #include <sys/os32.h>
2860011 #include <stddef.h>
2860012 #include <stdint.h>
2860013 #include <time.h>
2860014 //-----
2860015 #define CLOCK_FREQUENCY_DIVISOR \
2860016     (3579545/3/CLOCKS_PER_SEC) // [1]
2860017 //
2860018 // [1] Internal clock frequency is (3579545/3) Hz.
2860019 // This value is divided by
2860020 // CLOCK_FREQUENCY_DIVISOR, giving 100 Hz.
2860021 // The divisor value is fixed, because the code
2860022 // suppose that the clock frequency is 100 Hz!
2860023 //
2860024 //-----
2860025 #define PROC_EMPTY      0
2860026 #define PROC_CREATED   1
2860027 #define PROC_READY     2
2860028 #define PROC_RUNNING   3
2860029 #define PROC_SLEEPING  4
2860030 #define PROC_ZOMBIE    5
2860031 //-----
2860032 #define MAGIC_OS32_APPL 0x6F7333326170706CCLL // [2]
2860033 //
2860034 // [2] os32appl
2860035 //
2860036 //-----
2860037 #define PROCESS_MAX ((GDT_ITEMS/2)-1) // Process
2860038 // slots.
2860039 #define MAX_SIGNALS 31 // Max signal number.
2860040 //
2860041 typedef struct
2860042 {
2860043     pid_t ppid; // Parent PID.
2860044     pid_t pgrp; // Process group ID.
2860045     uid_t uid; // Real user ID
2860046     uid_t euid; // Effective user ID.
2860047     uid_t suid; // Saved user ID.
2860048     gid_t gid; // Real group ID
2860049     gid_t egid; // Effective group ID.
2860050     gid_t sgid; // Saved group ID.
2860051     dev_t device_tty; // Controlling terminal.
2860052     char path_cwd[PATH_MAX];
2860053     // Working directory path.
2860054     inode_t *inode_cwd; // Working directory inode.
2860055     int umask; // File creation mask.

```

```

2860056     unsigned long int sig_status; // Active signals.
2860057     unsigned long int sig_ignore; // Signals to be
2860058     // ignored.
2860059     uintptr_t sig_handler[MAX_SIGNALS]; // Opt. sig.
2860060     // handlers.
2860061     uintptr_t sig_handler_wrapper; // Special
2860062     // wrapper.
2860063     clock_t usage; // Clock ticks CPU time usage.
2860064     unsigned int status;
2860065     int wakeup_events; // Wake up for something.
2860066     int wakeup_signal; // Signal waited.
2860067     unsigned int wakeup_timer; // Seconds to wait
2860068     // for.
2860069     inode_t *wakeup_inode; // Inode waited.
2860070     sock_t *wakeup_sock; // Socket waited.
2860071     dev_t wakeup_dev; // Device waited.
2860072     addr_t address_text;
2860073     size_t domain_text;
2860074     addr_t address_data;
2860075     size_t domain_data;
2860076     size_t domain_stack; // Included inside the data.
2860077     size_t extra_data; // Extra data for 'brk()'.
2860078     uint32_t sp;
2860079     int ret;
2860080     char name[PATH_MAX];
2860081     fd_t fd[FOPEN_MAX];
2860082 } proc_t;
2860083 //
2860084 extern proc_t proc_table[PROCESS_MAX];
2860085 //
2860086 // ATTENTION: THERE IS NO WAY TO KEEP THE STACK DOMAIN
2860087 // IN A DIFFERENT ADDRESS SPACE THAN THE
2860088 // OTHER DATA. There is a simple explanation
2860089 // for such limitation: A POINTER TO DATA
2860090 // INSIDE THE STACK, CANNOT BE REACHED IF IT
2860091 // IS NOT INSIDE THE SAME ADDRESS SPACE!
2860092 //
2860093 // For the same reason, data or stack,
2860094 // cannot be moved (shifted) down, when more
2860095 // space is needed, because previous
2860096 // pointers would not work! So, to develop
2860097 // an extensible 'malloc' area, such memory
2860098 // is to be placed *after* the stack, moving
2860099 // *all* the data segment if necessary.
2860100 //-----
2860101 extern pid_t proc_current;
2860102 extern uint32_t proc_stack_pointer;
2860103 extern uint16_t proc_stack_segment_selector;
2860104 extern unsigned int proc_loops_per_clock;
2860105 //-----
2860106 typedef struct
2860107 {
2860108     uint32_t filler0;
2860109     uint64_t magic;
2860110     uint32_t data_offset;
2860111     uint32_t etext;
2860112     uint32_t edata;
2860113     uint32_t ebss;
2860114     uint32_t ssize;
2860115 } header_t;
2860116 //-----
2860117 typedef struct
2860118 {
2860119     uint32_t eax;
2860120     uint32_t ecx;
2860121     uint32_t edx;
2860122     uint32_t ebx;
2860123     uint32_t ebp;
2860124     uint32_t esi;
2860125     uint32_t edi;
2860126     uint32_t ds;
2860127     uint32_t es;
2860128     uint32_t fs;
2860129     uint32_t gs;
2860130     uint32_t eip;
2860131     uint32_t cs;
2860132     uint32_t eflags;
2860133 } stack_t;
2860134 //-----
2860135 void proc_init (void);
2860136 void proc_timer_init (clock_t freq);
2860137 void proc_delay (void);
2860138 void proc_scheduler (void);
2860139 void sysroutine (uint32_t syscallnr, uint32_t msg_off,
2860140                 uint32_t msg_size);
2860141 //-----
2860142 void *ptr (pid_t pid, void *p);

```

```

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

```

94.14.1	kernel/proc/proc_available.c	706
94.14.2	kernel/proc/proc_dump_memory.c	707
94.14.3	kernel/proc/proc_init.c	708
94.14.4	kernel/proc/proc_print.c	710
94.14.5	kernel/proc/proc_public.c	712
94.14.6	kernel/proc/proc_reference.c	712
94.14.7	kernel/proc/proc_sch_net.c	712
94.14.8	kernel/proc/proc_sch_signals.c	713
94.14.9	kernel/proc/proc_sch_terminals.c	714
94.14.10	kernel/proc/proc_sch_timers.c	719
94.14.11	kernel/proc/proc_scheduler.c	719
94.14.12	kernel/proc/proc_sig_chld.c	722
94.14.13	kernel/proc/proc_sig_cont.c	722
94.14.14	kernel/proc/proc_sig_core.c	723
94.14.15	kernel/proc/proc_sig_handler.c	724
94.14.16	kernel/proc/proc_sig_ignore.c	727
94.14.17	kernel/proc/proc_sig_off.c	727
94.14.18	kernel/proc/proc_sig_on.c	727
94.14.19	kernel/proc/proc_sig_status.c	727
94.14.20	kernel/proc/proc_sig_stop.c	727
94.14.21	kernel/proc/proc_sig_term.c	728
94.14.22	kernel/proc/proc_sys_exec.c	728
94.14.23	kernel/proc/proc_timer_init.c	738
94.14.24	kernel/proc/proc_wakeup_pipe_read.c	739
94.14.25	kernel/proc/proc_wakeup_pipe_write.c	739
94.14.26	kernel/proc/proc_wakeup_terminal.c	739
94.14.27	kernel/proc/ptr.c	740
94.14.28	kernel/proc/sysroutine.c	740

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             //

```

```

288052     if (size_written < 0)
288053     {
288054         s_close (pid, fdn);
288055         return;
288056     }
288057     }
288058     }
288059     s_close (pid, fdn);
288060 }

```

### 94.14.3 kernel/proc/proc\_init.c

« Si veda la sezione 93.20.3.

```

289001 #include <kernel/ibm_i386.h>
289002 #include <kernel/proc.h>
289003 #include <kernel/lib_k.h>
289004 #include <kernel/lib_s.h>
289005 #include <string.h>
289006 #include <kernel/multiboot.h>
289007 #include <kernel/dm.h>
289008 //-----
289009 extern uint32_t _k_start;
289010 extern uint32_t _k_end;
289011 extern uint32_t _k_text_end;
289012 extern uint32_t _k_data_end;
289013 extern uint32_t _k_bss_end;
289014 extern uint32_t _k_stack_top;
289015 extern uint32_t _k_stack_bottom;
289016 //-----
289017 void
289018 proc_init (void)
289019 {
289020     pid_t pid;
289021     int fdn; // File descriptor index;
289022     inode_t *inode;
289023     sb_t *sb;
289024     clock_t time_start;
289025     clock_t time_now;
289026     clock_t time_elapsed;
289027     unsigned long long int count;
289028     uintptr_t stack_top = (uintptr_t) &_k_stack_top;
289029     uintptr_t stack_bottom = (uintptr_t) &_k_stack_bottom;
289030     int sig;
289031     //
289032     // Set up the GDT table.
289033     //
289034     gdt ();
289035     //
289036     // Set up timer.
289037     //
289038     proc_timer_init (CLOCKS_PER_SEC);
289039     //
289040     // Disable external interrupt.
289041     //
289042     cli ();
289043     //
289044     // Set up the IDT table.
289045     //
289046     idt ();
289047     //
289048     // Set all memory reference to some invalid data.
289049     //
289050     for (pid = 0; pid < PROCESS_MAX; pid++)
289051     {
289052         proc_available (pid);
289053     }
289054     //
289055     // Set up the process table with the kernel.
289056     //
289057     proc_table[0].ppid = 0;
289058     proc_table[0].pgroup = 0;
289059     proc_table[0].uid = 0;
289060     proc_table[0].euid = 0;
289061     proc_table[0].suid = 0;
289062     proc_table[0].gid = 0;
289063     proc_table[0].egid = 0;
289064     proc_table[0].sgid = 0;
289065     proc_table[0].device_tty = DEV_UNDEFINED;
289066     proc_table[0].sig_status = 0;
289067     proc_table[0].sig_ignore = 0;
289068     proc_table[0].usage = 0;
289069     proc_table[0].status = PROC_RUNNING;
289070     proc_table[0].wakeuper_events = 0;
289071     proc_table[0].wakeuper_signal = 0;
289072     proc_table[0].wakeuper_timer = 0;
289073     proc_table[0].wakeuper_inode = NULL;

```

```

290074     proc_table[0].address_text = 0; // [1]
290075     proc_table[0].domain_text = (size_t) (&k_end); // [2]
290076     proc_table[0].address_data = 0; // [2]
290077     proc_table[0].domain_data = (size_t) 0; // [2]
290078     proc_table[0].domain_stack =
290079     (size_t) (stack_bottom - stack_top);
290080     proc_table[0].extra_data = (size_t) 0;
290081     proc_table[0].sp = 0; // [3]
290082     proc_table[0].ret = 0;
290083     proc_table[0].umask = 0022; // Default umask.
290084     strncpy (proc_table[0].path_cwd, "/", PATH_MAX);
290085     strncpy (proc_table[0].name, "os32 kernel", PATH_MAX);
290086     //
290087     // [1] The kernel text starts at 0x100000, that is,
290088     // 1 MiByte,
290089     // but the code expect to start at that address,
290090     // just like
290091     // the space from address 0 to 0xFFFFF is anyway
290092     // part
290093     // of the kernel. If the kernel is forked, as it
290094     // happens
290095     // when the 'init' process is to be run, it is
290096     // necessary
290097     // to consider part of the kernel all the addresses
290098     // starting
290099     // from zero.
290100     //
290101     // [2] The kernel is a unique block, where text and
290102     // data live
290103     // together.
290104     //
290105     // [3] The saved stack pointer location will be set
290106     // at the
290107     // next interrupt, or system call. That is why the
290108     // kernel
290109     // must send a null system call at the beginning of
290110     // its
290111     // work.
290112     //-----
290113     //
290114     // Ensure to have a terminated string.
290115     //
290116     proc_table[0].name[PATH_MAX - 1] = 0;
290117     //
290118     // Reset file descriptors.
290119     //
290120     for (fdn = 0; fdn < OPEN_MAX; fdn++)
290121     {
290122         proc_table[0].fd[fdn].fl_flags = 0;
290123         proc_table[0].fd[fdn].fd_flags = 0;
290124         proc_table[0].fd[fdn].file = NULL;
290125     }
290126     //
290127     // Reset 'sig_handler[]'.
290128     //
290129     for (sig = 0; sig < MAX_SIGNALS; sig++)
290130     {
290131         proc_table[0].sig_handler[sig] = (uintptr_t) NULL;
290132     }
290133     //
290134     // Allocate memory for the kernel.
290135     //
290136     // The BIOS data area (BDA) and extra BIOS at the
290137     // bottom
290138     // of 640 Kibyte, is already, formally, included
290139     // inside the
290140     // kernel.
290141     //
290142     // The allocation for data has no effect here,
290143     // because it is
290144     // the same as the text.
290145     //
290146     mb_alloc (proc_table[0].address_text,
290147             proc_table[0].domain_text);
290148     mb_alloc (proc_table[0].address_data,
290149             proc_table[0].domain_data);
290150     //
290151     // Enable and disable hardware interrupts (IRQ).
290152     //
290153     irq_on (0); // timer.
290154     irq_on (1); // keyboard
290155     irq_on (2); // PIC2: keep it ON! [4]
290156     irq_on (3); //
290157     irq_on (4); //
290158     irq_on (5); //
290159     irq_on (6); //
290160     irq_on (7); //

```



```

290009         / MEM_BLOCK_SIZE,
290010         (unsigned int) proc_table[pid].address_data
290011         / MEM_BLOCK_SIZE,
290012         (unsigned int) proc_table[pid].domain_data
290013         / MEM_BLOCK_SIZE,
290014         (unsigned int) proc_table[pid].sp,
290015         proc_table[pid].name);
290016     }
290017 }
290018 }
290019 }

```

#### 94.14.5 kernel/proc/proc\_public.c

« Si veda la sezione 93.20.5.

```

290001 #include <kernel/proc.h>
290002 //-----
290003 proc_t proc_table[PROCESS_MAX];
290004 pid_t proc_current = 0;
290005 uint16_t proc_stack_segment_selector = 24;
290006 uint32_t proc_stack_pointer;
290007 unsigned int proc_loops_per_clock;

```

#### 94.14.6 kernel/proc/proc\_reference.c

« Si veda la sezione 93.20.5.

```

290001 #include <kernel/proc.h>
290002 //-----
290003 proc_t *
290004 proc_reference (pid_t pid)
290005 {
290006     if (pid >= 0 && pid < PROCESS_MAX)
290007     {
290008         return (&proc_table[pid]);
290009     }
290010     else
290011     {
290012         return (NULL);
290013     }
290014 }

```

#### 94.14.7 kernel/proc/proc\_sch\_net.c

« Si veda la sezione 93.20.6.

```

290001 #include <kernel/proc.h>
290002 #include <kernel/lib_k.h>
290003 #include <kernel/lib_s.h>
290004 #include <kernel/driver/nic/ne2k.h>
290005 #include <kernel/net.h>
290006 #include <kernel/driver/pci.h>
290007 //-----
290008 #define DEBUG 0
290009 //-----
290010 void
290011 proc_sch_net (void)
290012 {
290013     int n;          // NET table index.
290014     int counter = 0;
290015     int pid;
290016     int tcp_wake_up;
290017     //
290018     // Do what there is to do with network interfaces,
290019     // in particular get
290020     // received frames, into the
290021     // 'net_table[n]...buffer[f]' table.
290022     //
290023     for (n = 0; n < NET_MAX_DEVICES; n++)
290024     {
290025         if (net_table[n].type == NET_DEV_ETH_NE2K)
290026         {
290027             //
290028             // The function 'ne2k_isr()' will call also
290029             // 'ne2k_rx()'
290030             // that is responsible for placing Ethernet
290031             // frames inside
290032             // 'ethernet_table[eth].frame_info[f]'.
290033             //
290034             ne2k_isr (net_table[n].ethernet.base_io);
290035         }
290036         else if (net_table[n].type == NET_DEV_LOOPBACK)
290037         {
290038             //
290039             // Packets should be already inside the
290040             // packet buffer table.

```

```

290041         //
290042         ;
290043     }
290044 }
290045 //
290046 // Do something with received Ethernet frames.
290047 //
290048 counter = net_rx ();
290049 //
290050 if (DEBUG)
290051 {
290052     if (counter)
290053     {
290054         k_printf ("%i packet(s) received\n", counter);
290055     }
290056 }
290057 //
290058 // Do something with TCP connections.
290059 //
290060 tcp_wake_up = tcp ();
290061 //
290062 // Wake up sleeping processes.
290063 //
290064 if (counter || tcp_wake_up == TCP_TRY_READ)
290065 {
290066     //
290067     // Wake up sleeping processes, waiting to read
290068     // from a socket.
290069     //
290070     if (DEBUG)
290071     {
290072         k_printf
290073             ("wake up processes, waiting "
290074              "for a socket!\n");
290075     }
290076     for (pid = 1; pid < PROCESS_MAX; pid++)
290077     {
290078         if (proc_table[pid].status == PROC_SLEEPING
290079             && (proc_table[pid].wakeup_events
290080                 & WAKEUP_EVENT_SOCKET_READ))
290081         {
290082             proc_table[pid].wakeup_events = 0;
290083             proc_table[pid].status = PROC_READY;
290084         }
290085     }
290086 }
290087 //
290088 if (tcp_wake_up == TCP_TRY_WRITE)
290089 {
290090     //
290091     // Wake up sleeping processes, waiting to write
290092     // to a socket.
290093     //
290094     for (pid = 1; pid < PROCESS_MAX; pid++)
290095     {
290096         if (proc_table[pid].status == PROC_SLEEPING
290097             && (proc_table[pid].wakeup_events
290098                 & WAKEUP_EVENT_SOCKET_WRITE))
290099         {
290100             proc_table[pid].wakeup_events = 0;
290101             proc_table[pid].status = PROC_READY;
290102         }
290103     }
290104 }
290105 }

```

#### 94.14.8 kernel/proc/proc\_sch\_signals.c

« Si veda la sezione 93.20.7.

```

294001 #include <kernel/proc.h>
294002 //-----
294003 void
294004 proc_sch_signals (void)
294005 {
294006     pid_t pid;
294007     //
294008     // The PID scan starts from 1: you will not send
294009     // signals to the
294010     // kernel!
294011     //
294012     for (pid = 1; pid < PROCESS_MAX; pid++)
294013     {
294014         proc_sig_term (pid, SIGHUP);
294015         proc_sig_term (pid, SIGINT);
294016         proc_sig_core (pid, SIGQUIT);
294017         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                                 {

```



```

290153 //
290154 // 'pid_sub' is a
290155 // son of
290156 // 'pid' and is part
290157 // of the
290158 // same process
290159 // group. 'pid_sub'
290160 // is candidate for
290161 // kill.
290162 //
290163 break;
290164 }
290165 }
290166 //
290167 if (pid_sub >= PROCESS_MAX)
290168 {
290169 //
290170 // There is no son for
290171 // 'pid', sorry.
290172 //
290173 s_kill ((pid_t) 0, pid,
290174 SIGINT);
290175 //
290176 // No more scan.
290177 //
290178 break;
290179 }
290180 }
290181 }
290182 }
290183 //
290184 // Just reset input line and return.
290185 //
290186 tty->status = TTY_INPUT_LINE_EDITING;
290187 tty->lpr = 0;
290188 tty->lpw = 0;
290189 tty->line[0] = 0;
290190 //
290191 return;
290192 }
290193 //
290194 // Replace the INTR character with zero.
290195 //
290196 key = 0;
290197 }
290198 }
290199 //
290200 // Check if something is to be ignored.
290201 //
290202 if (key == '\r' && (tty->attr.c_iflag & IGNCR))
290203 {
290204 return;
290205 }
290206 //
290207 // Check if something is to be replaced, before
290208 // editing.
290209 //
290210 if (key == '\n' && (tty->attr.c_iflag & INLCR))
290211 {
290212 key = '\r';
290213 }
290214 //
290215 if (key == '\r' && (tty->attr.c_iflag & ICRNL))
290216 {
290217 key = '\n';
290218 }
290219 //
290220 // Edit the canonical input line.
290221 // Input is accepted only if status is ok.
290222 //
290223 if (tty->status == TTY_INPUT_LINE_EDITING)
290224 {
290225 //
290226 // Fix internal line positions.
290227 //
290228 if (tty->lpw < 0)
290229 {
290230 tty->lpw = 0;
290231 }
290232 if (tty->lpw >= MAX_CANON)
290233 {
290234 tty->lpw = MAX_CANON - 1;
290235 overflow = 1; // Too much input!
290236 }
290237 if (tty->lpr < 0)
290238 {
290239 tty->lpr = 0;

```

```

290240 }
290241 if (tty->lpr > tty->lpw)
290242 {
290243 tty->lpr = tty->lpw;
290244 }
290245 //
290246 // Check input key.
290247 //
290248 if (key == '\0')
290249 {
290250 //
290251 // A INTR replaced into zero.
290252 //
290253 tty->line[tty->lpw] = key;
290254 tty->status = TTY_INPUT_LINE_CLOSED;
290255 proc_wakeup_terminal ();
290256 }
290257 else if (key == '\n')
290258 {
290259 tty->line[tty->lpw] = key;
290260 tty->status = TTY_INPUT_LINE_CLOSED;
290261 proc_wakeup_terminal ();
290262 }
290263 else if (tty->attr.c_cc[VEOF]
290264 && key == tty->attr.c_cc[VEOF])
290265 {
290266 //
290267 // EOF is not included inside the line: just
290268 // replace the
290269 // with zero.
290270 //
290271 key = 0;
290272 tty->line[tty->lpw] = key;
290273 tty->status = TTY_INPUT_LINE_CLOSED;
290274 proc_wakeup_terminal ();
290275 }
290276 else if (tty->attr.c_cc[VEOL]
290277 && key == tty->attr.c_cc[VEOL])
290278 {
290279 tty->line[tty->lpw] = key;
290280 tty->status = TTY_INPUT_LINE_CLOSED;
290281 proc_wakeup_terminal ();
290282 }
290283 else if (tty->attr.c_cc[VERASE]
290284 && key == tty->attr.c_cc[VERASE])
290285 {
290286 //
290287 // Save how many characters to be killed, if
290288 // echo is
290289 // enabled.
290290 //
290291 if (overflow)
290292 {
290293 k = tty->lpw + 1;
290294 //
290295 // The 'tty->lpw' was already reduced.
290296 //
290297 }
290298 else
290299 {
290300 k = tty->lpw;
290301 //
290302 // Reduce write index: if it is less
290303 // than zero, it will
290304 // be fixed.
290305 //
290306 tty->lpw--;
290307 }
290308 }
290309 else if (tty->attr.c_cc[VKILL]
290310 && key == tty->attr.c_cc[VKILL])
290311 {
290312 //
290313 // Save how many characters to be killed, if
290314 // echo is
290315 // enabled.
290316 //
290317 if (overflow)
290318 {
290319 k = tty->lpw + 1;
290320 }
290321 else
290322 {
290323 k = tty->lpw;
290324 }
290325 //
290326 // Reset input line.

```

```

290027 //
290028 tty->lpw = 0;
290029 tty->lpr = 0;
290030 tty->line[0] = 0;
290031 }
290032 else
290033 {
290034     tty->line[tty->lpw] = key;
290035     tty->lpw++;
290036 }
290037 //
290038 // Echo.
290039 //
290040 if (!(tty->attr.c_lflag & ECHO))
290041 {
290042     //
290043     // No echo. But NL might be echoed anyway.
290044     //
290045     if (key == '\n' && (tty->attr.c_lflag & ECHONL))
290046     {
290047         dev_io ((pid_t) 0, tty->device,
290048                DEV_WRITE, (off_t) 0, &key,
290049                (size_t) 1, NULL);
290050     }
290051     //
290052     return;
290053 }
290054 //
290055 // The echo is requested.
290056 //
290057 if (key == 0)
290058 {
290059     //
290060     // There is nothing to echo.
290061     //
290062     ;
290063 }
290064 else if (tty->attr.c_cc[VERASE]
290065         && key == tty->attr.c_cc[VERASE])
290066 {
290067     if (tty->attr.c_lflag & ECHOE)
290068     {
290069         if (k > 0)
290070         {
290071             dev_io ((pid_t) 0, tty->device,
290072                    DEV_WRITE, (off_t) 0,
290073                    "\b \b", (size_t) 3, NULL);
290074         }
290075     }
290076     else
290077     {
290078         dev_io ((pid_t) 0, tty->device,
290079                DEV_WRITE, (off_t) 0, &key,
290080                (size_t) 1, NULL);
290081     }
290082 }
290083 else if (tty->attr.c_cc[VKILL]
290084         && key == tty->attr.c_cc[VKILL])
290085 {
290086     if (tty->attr.c_lflag & ECHOK)
290087     {
290088         for (; k > 0; k--)
290089         {
290090             dev_io ((pid_t) 0, tty->device,
290091                    DEV_WRITE, (off_t) 0,
290092                    "\b \b", (size_t) 3, NULL);
290093         }
290094     }
290095 }
290096 else if (key == '\n')
290097 {
290098     if (tty->attr.c_lflag & ECHONL)
290099     {
290100         dev_io ((pid_t) 0, tty->device,
290101                DEV_WRITE, (off_t) 0, &key,
290102                (size_t) 1, NULL);
290103     }
290104 }
290105 else
290106 {
290107     //
290108     // If there was an overflow, the last
290109     // character is
290110     // overwriting the last position, so a back
290111     // space
290112     // is printed.
290113     //

```

```

290414         if (overflow)
290415         {
290416             dev_io ((pid_t) 0, tty->device,
290417                    DEV_WRITE, (off_t) 0, "\b",
290418                    (size_t) 1, NULL);
290419         }
290420         //
290421         // Now show the input character.
290422         //
290423         dev_io ((pid_t) 0, tty->device, DEV_WRITE,
290424                (off_t) 0, &key, (size_t) 1, NULL);
290425     }
290426 }
290427 }

```

#### 94.14.10 kernel/proc/proc\_sch\_timers.c

Si veda la sezione 93.20.9.

```

296001 #include <kernel/proc.h>
296002 #include <kernel/lib_k.h>
296003 #include <kernel/lib_s.h>
296004 //-----
296005 void
296006 proc_sch_timers (void)
296007 {
296008     static unsigned long long int previous_time;
296009     unsigned long long int current_time;
296010     unsigned int pid;
296011     current_time = s_time ((pid_t) 0, NULL);
296012     if (previous_time != current_time)
296013     {
296014         for (pid = 0; pid < PROCESS_MAX; pid++)
296015         {
296016             if ((proc_table[pid].wakeup_events &
296017                 WAKEUP_EVENT_TIMER)
296018                 && (proc_table[pid].status == PROC_SLEEPING)
296019                 && (proc_table[pid].wakeup_timer > 0))
296020             {
296021                 proc_table[pid].wakeup_timer--;
296022                 if (proc_table[pid].wakeup_timer == 0)
296023                 {
296024                     proc_table[pid].status = PROC_READY;
296025                 }
296026             }
296027         }
296028     }
296029     previous_time = current_time;
296030 }

```

#### 94.14.11 kernel/proc/proc\_scheduler.c

Si veda la sezione 93.20.10.

```

297001 #include <kernel/proc.h>
297002 #include <kernel/lib_k.h>
297003 #include <kernel/lib_s.h>
297004 #include <kernel/net.h>
297005 #include <stdint.h>
297006 //-----
297007 #define DEBUG 1
297008 //-----
297009 extern uint32_t _ksp;
297010 extern uint32_t proc_stack_pointer;
297011 extern uint16_t proc_stack_segment_selector;
297012 extern pid_t proc_current;
297013 //-----
297014 void
297015 proc_scheduler (void)
297016 {
297017     pid_t prev;
297018     pid_t next;
297019     addr_t stack_top;
297020     addr_t stack_bottom;
297021     uint32_t saved_stack_pointer = proc_stack_pointer;
297022     //
297023     static unsigned long long int previous_clock;
297024     unsigned long long int current_clock;
297025     //
297026     // Check the current stack size.
297027     //
297028     if (proc_table[proc_current].domain_data == 0)
297029     {
297030         stack_bottom = proc_table[proc_current].domain_text;
297031     }
297032     else
297033     {

```

```

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     }
2980042     else
2980043     {
2980044         //
2980045         // All other cases, means to remove all dead
2980046         // children.
2980047         //
2980048         for (child = 1; child < PROCESS_MAX; child++)
2980049         {
2980050             if (proc_table[child].ppid == parent
2980051                 && proc_table[child].status ==
2980052                     PROC_ZOMBIE)
2980053             {
2980054                 proc_available (child);
2980055             }
2980056         }
2980057     }
2980058     proc_sig_off (parent, sig);
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     }
2990016     else
2990017     {
2990018         if (proc_table[pid].sig_handler[sig] !=
2990019             (uintptr_t) NULL)
2990020         {
2990021             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     }
3000020     else
3000021     {
3000022         if (proc_table[pid].sig_handler[sig] !=
3000023             (uintptr_t) NULL)
3000024         {
3000025             proc_sig_handler (pid, sig);
3000026         }
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;

```

```

300173 addr_stack_pointer -= 12; // Three more
300174 // elements.
300175 new = (void *) addr_stack_pointer;
300176 //
300177 // Verify if the code segment was correctly found.
300178 //
300179 if (DEBUG)
300180 {
300181     k_printf
300182     ("[%s] EAX:%04x ECX:%04x EDX:%04x "
300183      "EBX:%04x EBP:%04x "
300184      "ESI:%04x EDI:%04x DS:%04x ES:%04x "
300185      "FS:%04x GS:%04x "
300186      "EIP:%04x CS:%04x EFLAGS:%04x\n", __FILE__,
300187      (int) old->eax, (int) old->ecx,
300188      (int) old->edx, (int) old->ebx,
300189      (int) old->ebp, (int) old->esi,
300190      (int) old->edi, (int) old->ds, (int) old->es,
300191      (int) old->fs, (int) old->gs, (int) old->eip,
300192      (int) old->cs, (int) old->eflags);
300193 }
300194 //
300195 // Move data, to arrange the new stack. The order
300196 // does
300197 // matter, as the new frame overwrites the old one.
300198 //
300199 new->eax = old->eax;
300200 new->ecx = old->ecx;
300201 new->edx = old->edx;
300202 new->ebx = old->ebx;
300203 new->ebp = old->ebp;
300204 new->esi = old->esi;
300205 new->edi = old->edi;
300206 new->ds = old->ds;
300207 new->es = old->es;
300208 new->fs = old->fs;
300209 new->gs = old->gs;
300210 //
300211 old_eflags = old->eflags;
300212 old_cs = old->cs;
300213 old_eip = old->eip;
300214 //
300215 new->wrapper = proc_table[pid].sig_handler_wrapper;
300216 new->cs = old_cs;
300217 new->eflags = old_eflags;
300218 new->handler = proc_table[pid].sig_handler[sig];
300219 new->signal = sig;
300220 new->eip = old_eip;
300221 //
300222 // Tell what is changed inside the stack.
300223 //
300224 if (DEBUG)
300225 {
300226     k_printf
300227     ("[%s] EAX:%04x ECX:%04x EDX:%04x "
300228      "EBX:%04x EBP:%04x "
300229      "ESI:%04x EDI:%04x DS:%04x ES:%04x "
300230      "FS:%04x GS:%04x "
300231      "wrapper:%04x CS:%04x EFLAGS:%04x "
300232      "handler:%04x "
300233      "signal:%04x EIP:%04x\n", __FILE__,
300234      (int) new->eax, (int) new->ecx,
300235      (int) new->edx, (int) new->ebx,
300236      (int) new->ebp, (int) new->esi,
300237      (int) new->edi, (int) new->ds, (int) new->es,
300238      (int) new->fs, (int) new->gs,
300239      (int) new->wrapper, (int) new->cs,
300240      (int) new->eflags, (int) new->handler,
300241      (int) new->signal, (int) new->eip);
300242 }
300243 //
300244 // Change the stack pointer of the process, as it
300245 // was increased.
300246 //
300247 proc_table[pid].sp = addr_stack_pointer - addr_data_top;
300248 //
300249 // Reset the signal handler, as in traditional Unix,
300250 // with
300251 // all the consequences that such implementation
300252 // will give.
300253 //
300254 proc_table[pid].sig_handler[sig] = (uintptr_t) NULL;
300255 proc_sig_off (pid, sig);
300256 //
300257 // Wake up the process if it is sleeping.
300258 //
300259 proc_table[pid].wakeup_events = 0;

```

```

300260 proc_table[pid].status = PROC_READY;
300261 }

```

#### 94.14.16 kernel/proc/proc\_sig\_ignore.c

Si veda la sezione [93.20.15](#).

```

302001 #include <kernel/proc.h>
302002 //-----
302003 int
302004 proc_sig_ignore (pid_t pid, int sig)
302005 {
302006     unsigned long int flag = 1L << (sig - 1);
302007     if (proc_table[pid].sig_ignore & flag)
302008     {
302009         return (1);
302010     }
302011     else
302012     {
302013         return (0);
302014     }
302015 }

```

#### 94.14.17 kernel/proc/proc\_sig\_off.c

Si veda la sezione [93.20.17](#).

```

303001 #include <kernel/proc.h>
303002 //-----
303003 void
303004 proc_sig_off (pid_t pid, int sig)
303005 {
303006     unsigned long int flag = 1L << (sig - 1);
303007     proc_table[pid].sig_status ^= flag;
303008 }

```

#### 94.14.18 kernel/proc/proc\_sig\_on.c

Si veda la sezione [93.20.17](#).

```

304001 #include <kernel/proc.h>
304002 //-----
304003 void
304004 proc_sig_on (pid_t pid, int sig)
304005 {
304006     unsigned long int flag = 1L << (sig - 1);
304007     proc_table[pid].sig_status |= flag;
304008 }

```

#### 94.14.19 kernel/proc/proc\_sig\_status.c

Si veda la sezione [93.20.18](#).

```

305001 #include <kernel/proc.h>
305002 //-----
305003 int
305004 proc_sig_status (pid_t pid, int sig)
305005 {
305006     unsigned long int flag = 1L << (sig - 1);
305007     if (proc_table[pid].sig_status & flag)
305008     {
305009         return (1);
305010     }
305011     else
305012     {
305013         return (0);
305014     }
305015 }

```

#### 94.14.20 kernel/proc/proc\_sig\_stop.c

Si veda la sezione [93.20.19](#).

```

306001 #include <kernel/proc.h>
306002 //-----
306003 void
306004 proc_sig_stop (pid_t pid, int sig)
306005 {
306006     if (proc_sig_status (pid, sig))
306007     {
306008         if (proc_sig_ignore (pid, sig) && !(sig == SIGSTOP))
306009         {
306010             proc_sig_off (pid, sig);
306011         }
306012     }
306013     else
306014     {

```

```

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 (EACCESS); // 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),

```

```

308464         1, 1, 0);
308465 //
308466 // Update process DATA segment inside the GDT table.
308467 //
308468 if (process_domain_data > 0)
308469 {
308470     gdt_segment (gdt_pid_to_segment_data (pid),
308471                 (uint32_t) allocated_data,
308472                 (uint32_t) (process_domain_data /
308473                             4096), 1, 0, 0);
308474 }
308475 else
308476 {
308477     gdt_segment (gdt_pid_to_segment_data (pid),
308478                 (uint32_t) allocated_text,
308479                 (uint32_t) (process_domain_text /
308480                             4096), 1, 0, 0);
308481 }
308482 //
308483 // Calculate segment descriptors.
308484 //
308485 segment_text = (gdt_pid_to_segment_text (pid) << 3) + 0;
308486 segment_data = (gdt_pid_to_segment_data (pid) << 3) + 0;
308487 //
308488 // Where is the stack?
308489 //
308490 if (process_domain_data > 0)
308491 {
308492     stack_location = allocated_data;
308493 }
308494 else
308495 {
308496     stack_location = allocated_text;
308497 }
308498 //
308499 // Put environment data inside the stack.
308500 //
308501 new_sp -= env_data_size; // -----
308502 // ENVIRONMENT
308503 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308504         (off_t) (stack_location + new_sp),
308505         env_data, env_data_size, NULL);
308506 //
308507 // Put arguments data inside the stack.
308508 //
308509 new_sp -= arg_data_size; // -----
308510 // ARGUMENTS
308511 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308512         (off_t) (stack_location + new_sp),
308513         arg_data, arg_data_size, NULL);
308514 //
308515 // Put envp[] inside the stack, updating all the
308516 // pointers.
308517 //
308518 new_sp -= 4; // ----- NULL
308519 stack_element = (uint32_t) NULL;
308520 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308521         (off_t) (stack_location + new_sp),
308522         &stack_element, (sizeof stack_element), NULL);
308523 //
308524 // Calculate memory pointers from original relative
308525 // pointers, inside the environment array of
308526 // pointers.
308527 //
308528 p_off = new_sp;
308529 p_off += 4;
308530 p_off += arg_data_size;
308531 for (i = 0; i < envc; i++)
308532 {
308533     envp[i] += p_off;
308534 }
308535 //
308536 new_sp -= (envc * (sizeof (char *))); // ---- *envp[]
308537 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308538         (off_t) (stack_location + new_sp),
308539         envp, (envc * (sizeof (char *))), NULL);
308540 //
308541 // Save the envp[] location, needed in the
308542 // following.
308543 //
308544 envp_address = new_sp;
308545 //
308546 // Put argv[] inside the stack, updating all the
308547 // pointers.
308548 //
308549 new_sp -= 4; // ----- NULL
308550 stack_element = (uint32_t) NULL;

```

```

308551 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308552         (off_t) (stack_location + new_sp),
308553         &stack_element, (sizeof stack_element), NULL);
308554 //
308555 // Calculate memory pointers from original relative
308556 // pointers, inside the arguments array of pointers.
308557 //
308558 p_off = new_sp;
308559 p_off += 4;
308560 p_off += (envc * (sizeof (char *)));
308561 p_off += 4;
308562 for (i = 0; i < argc; i++)
308563 {
308564     argv[i] += p_off;
308565 }
308566 //
308567 new_sp -= (argc * (sizeof (char *))); // ---- *argv[]
308568 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308569         (off_t) (stack_location + new_sp),
308570         argv, (argc * (sizeof (char *))), NULL);
308571 //
308572 // Save the argv[] location, needed in the
308573 // following.
308574 //
308575 argv_address = new_sp;
308576 //
308577 // Put the pointer to the array envp[].
308578 //
308579 new_sp -= 4; // ----- argc
308580 stack_element = envp_address;
308581 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308582         (off_t) (stack_location + new_sp),
308583         &stack_element, (sizeof stack_element), NULL);
308584 //
308585 // Put the pointer to the array argv[].
308586 //
308587 new_sp -= 4; // ----- argc
308588 stack_element = argv_address;
308589 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308590         (off_t) (stack_location + new_sp),
308591         &stack_element, (sizeof stack_element), NULL);
308592 //
308593 // Put argc inside the stack.
308594 //
308595 new_sp -= 4; // ----- argc
308596 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308597         (off_t) (stack_location + new_sp),
308598         &argc, (sizeof argc), NULL);
308599 //
308600 // Set the rest of the stack.
308601 //
308602 new_sp -= 4; // ----- EFLAGS
308603 stack_element = 0x0200;
308604 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308605         (off_t) (stack_location + new_sp),
308606         &stack_element, (sizeof stack_element), NULL);
308607 new_sp -= 4; // ----- CS
308608 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308609         (off_t) (stack_location + new_sp),
308610         &segment_text, (sizeof segment_text), NULL);
308611 new_sp -= 4; // ----- EIP
308612 stack_element = 0;
308613 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308614         (off_t) (stack_location + new_sp),
308615         &stack_element, (sizeof stack_element), NULL);
308616 new_sp -= 4; // ----- GS
308617 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308618         (off_t) (stack_location + new_sp),
308619         &segment_data, (sizeof segment_data), NULL);
308620 new_sp -= 4; // ----- FS
308621 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308622         (off_t) (stack_location + new_sp),
308623         &segment_data, (sizeof segment_data), NULL);
308624 new_sp -= 4; // ----- ES
308625 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308626         (off_t) (stack_location + new_sp),
308627         &segment_data, (sizeof segment_data), NULL);
308628 new_sp -= 4; // ----- DS
308629 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308630         (off_t) (stack_location + new_sp),
308631         &segment_data, (sizeof segment_data), NULL);
308632 new_sp -= 4; // ----- EDI
308633 stack_element = 0;
308634 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308635         (off_t) (stack_location + new_sp),
308636         &stack_element, (sizeof stack_element), NULL);
308637 new_sp -= 4; // ----- ESI

```

```

308638 stack_element = 0;
308639 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308640         (off_t) (stack_location + new_sp),
308641         &stack_element, (sizeof stack_element), NULL);
308642 new_sp -= 4; // ----- EBP
308643 stack_element = 0;
308644 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308645         (off_t) (stack_location + new_sp),
308646         &stack_element, (sizeof stack_element), NULL);
308647 new_sp -= 4; // ----- EBX
308648 stack_element = 0;
308649 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308650         (off_t) (stack_location + new_sp),
308651         &stack_element, (sizeof stack_element), NULL);
308652 new_sp -= 4; // ----- EDX
308653 stack_element = 0;
308654 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308655         (off_t) (stack_location + new_sp),
308656         &stack_element, (sizeof stack_element), NULL);
308657 new_sp -= 4; // ----- ECX
308658 stack_element = 0;
308659 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308660         (off_t) (stack_location + new_sp),
308661         &stack_element, (sizeof stack_element), NULL);
308662 new_sp -= 4; // ----- EAX
308663 stack_element = 0;
308664 dev_io ((pid_t) 0, DEV_MEM, DEV_WRITE,
308665         (off_t) (stack_location + new_sp),
308666         &stack_element, (sizeof stack_element), NULL);
308667 //
308668 // Close process file descriptors, if the
308669 // 'FD_CLOEXEC' flag
308670 // is present.
308671 //
308672 for (fdn = 0; fdn < OPEN_MAX; fdn++)
308673 {
308674     if (proc_table[pid].fd[0].file != NULL)
308675     {
308676         if (proc_table[pid].fd[0].fd_flags & FD_CLOEXEC)
308677         {
308678             s_close (pid, fdn);
308679         }
308680     }
308681 }
308682 //
308683 // Select device for standard I/O, if a standard I/O
308684 // stream must be
308685 // opened.
308686 //
308687 if (proc_table[pid].device_tty != 0)
308688 {
308689     device = proc_table[pid].device_tty;
308690 }
308691 else
308692 {
308693     device = DEV_TTY;
308694 }
308695 //
308696 // Prepare missing standard file descriptors. The
308697 // function
308698 // 'file_stdio_dev_make()' arranges the value for
308699 // 'errno' if
308700 // necessary. If a standard file descriptor cannot
308701 // be allocated,
308702 // the program is left without it.
308703 //
308704 if (proc_table[pid].fd[0].file == NULL)
308705 {
308706     file =
308707     file_stdio_dev_make (device, S_IFCHR, O_RDONLY);
308708     if (file != NULL) // stdin
308709     {
308710         proc_table[pid].fd[0].fl_flags = O_RDONLY;
308711         proc_table[pid].fd[0].fd_flags = 0;
308712         proc_table[pid].fd[0].file = file;
308713         proc_table[pid].fd[0].file->offset = 0;
308714     }
308715 }
308716 if (proc_table[pid].fd[1].file == NULL)
308717 {
308718     file =
308719     file_stdio_dev_make (device, S_IFCHR, O_WRONLY);
308720     if (file != NULL) // stdout
308721     {
308722         proc_table[pid].fd[1].fl_flags = O_WRONLY;
308723         proc_table[pid].fd[1].fd_flags = 0;
308724         proc_table[pid].fd[1].file = file;

```

```

308725         proc_table[pid].fd[1].file->offset = 0;
308726     }
308727 }
308728 if (proc_table[pid].fd[2].file == NULL)
308729 {
308730     file =
308731     file_stdio_dev_make (device, S_IFCHR, O_WRONLY);
308732     if (file != NULL) // stderr
308733     {
308734         proc_table[pid].fd[2].fl_flags = O_WRONLY;
308735         proc_table[pid].fd[2].fd_flags = 0;
308736         proc_table[pid].fd[2].file = file;
308737         proc_table[pid].fd[2].file->offset = 0;
308738     }
308739 }
308740 //
308741 // Prepare to switch
308742 //
308743 previous_address_text = proc_table[pid].address_text;
308744 previous_domain_text = proc_table[pid].domain_text;
308745 previous_address_data = proc_table[pid].address_data;
308746 previous_domain_data = proc_table[pid].domain_data;
308747 previous_domain_stack = proc_table[pid].domain_stack;
308748 previous_extra_data = proc_table[pid].extra_data;
308749 //
308750 proc_table[pid].address_text = allocated_text;
308751 proc_table[pid].domain_text = process_domain_text;
308752 proc_table[pid].address_data = allocated_data;
308753 proc_table[pid].domain_data = process_domain_data;
308754 proc_table[pid].domain_stack = process_domain_stack;
308755 proc_table[pid].extra_data = (size_t) 0;
308756 proc_table[pid].sp = new_sp;
308757 strncpy (proc_table[pid].name, path, PATH_MAX);
308758 //
308759 // Ensure to have a terminated string.
308760 //
308761 proc_table[pid].name[PATH_MAX - 1] = 0;
308762 //
308763 // Reset 'sig_handler[]'.
308764 //
308765 for (sig = 0; sig < MAX_SIGNALS; sig++)
308766 {
308767     proc_table[pid].sig_handler[sig] = (uintptr_t) NULL;
308768 }
308769 //
308770 // Free previous data memory (included stack).
308771 //
308772 if (previous_domain_data > 0)
308773 {
308774     mb_free (previous_address_data,
308775             previous_domain_data + previous_extra_data);
308776     if (DEBUG)
308777     {
308778         k_printf ("%s:%i:mb_free(%i, %zi)",
308779                 __FILE__, __LINE__,
308780                 (unsigned int)
308781                 previous_address_data,
308782                 previous_domain_data +
308783                 previous_extra_data);
308784     }
308785 }
308786 //
308787 // Free code memory if not shared.
308788 //
308789 for (proc_count = 0, extra = 0; extra < PROCESS_MAX;
308790      extra++)
308791 {
308792     if (proc_table[extra].status == PROC_EMPTY ||
308793         proc_table[extra].status == PROC_ZOMBIE)
308794     {
308795         continue;
308796     }
308797     if (previous_address_text ==
308798         proc_table[extra].address_text)
308799     {
308800         proc_count++;
308801     }
308802 }
308803 if (proc_count == 0)
308804 {
308805     //
308806     // The code segment can be released, because no
308807     // other
308808     // process is using it.
308809     //
308810     if (previous_domain_data > 0)
308811     {

```

```

308812     mb_free (previous_address_text,
308813             previous_domain_text +
308814             previous_extra_data);
308815     if (DEBUG)
308816     {
308817         k_printf ("%s:%i:mb_free(%i, %zi)",
308818                 __FILE__, __LINE__,
308819                 (unsigned int)
308820                 previous_address_text,
308821                 previous_domain_text +
308822                 previous_extra_data);
308823     }
308824 }
308825 else
308826 {
308827     mb_free (previous_address_text,
308828             previous_domain_text);
308829     if (DEBUG)
308830     {
308831         k_printf ("%s:%i:mb_free(%i, %zi)",
308832                 __FILE__, __LINE__,
308833                 (unsigned int)
308834                 previous_address_text,
308835                 previous_domain_text);
308836     }
308837 }
308838 }
308839 //
308840 // Change the segment and the stack pointer, from
308841 // the interrupt.
308842 // [1] Anyway, the stack segment selector does not
308843 // change.
308844 //
308845 proc_stack_segment_selector = segment_data; // [1]
308846 proc_stack_pointer = proc_table[pid].sp;
308847 //
308848 //
308849 //
308850 return (0);
308851 }

```

#### 94.14.23 kernel/proc/proc\_timer\_init.c

« Si veda la sezione 93.20.22.

```

309001 #include <kernel/proc.h>
309002 #include <stdint.h>
309003 #include <kernel/lib_k.h>
309004 #include <kernel/ibm_i386.h>
309005 #include <stdint.h>
309006 //-----
309007 void
309008 proc_timer_init (clock_t freq)
309009 {
309010     int input_freq = 1193180;
309011     //
309012     // La frequenza di riferimento è 1,19318 MHz, la
309013     // quale va
309014     // divisa per la frequenza che si intende avere
309015     // effettivamente.
309016     //
309017     int divisor = input_freq / freq;
309018     //
309019     // Il risultato deve essere un valore intero
309020     // maggiore di zero
309021     // e inferiore di UINT16_MAX, altrimenti è stata
309022     // chiesta una
309023     // frequenza troppo elevata o troppo bassa.
309024     //
309025     if (divisor == 0 || divisor > UINT16_MAX)
309026     {
309027         k_printf
309028         ("[%s] ERROR: IRQ 0 frequency wrong: %i Hz!\n"
309029          "[%s] The min allowed frequency \n"
309030          "[%s] is 18.22 Hz.\n",
309031          "[%s] The max allowed frequency \n"
309032          "[%s] is 1.19 MHz.\n",
309033          __func__, freq, __func__, __func__, __func__,
309034          __func__);
309035     }
309036     return;
309037 //
309038 // Il valore che si ottiene, ovvero il «divisore»,
309039 // va
309040 // comunicato al PIT (programmable interval timer),
309041 // spezzandolo in due parti.
309042 //

```

```

309043     out_8 ((uint32_t) 0x43, (uint32_t) 0x36);
309044     //
309045     // Lower byte.
309046     //
309047     out_8 ((uint32_t) 0x40, (uint32_t) (divisor & 0x0F));
309048     //
309049     // Higher byte.
309050     //
309051     out_8 ((uint32_t) 0x40, (uint32_t) (divisor / 0x10));
309052 }

```

#### 94.14.24 kernel/proc/proc\_wakeup\_pipe\_read.c

« Si veda la sezione 93.20.23.

```

310001 #include <kernel/proc.h>
310002 //-----
310003 void
310004 proc_wakeup_pipe_read (inode_t * inode)
310005 {
310006     pid_t pid;
310007
310008     for (pid = 1; pid < PROCESS_MAX; pid++)
310009     {
310010         if ((proc_table[pid].status == PROC_SLEEPING)
310011             && (proc_table[pid].wakeup_events
310012                & WAKEUP_EVENT_PIPE_READ)
310013             && (proc_table[pid].wakeup_inode == inode))
310014         {
310015             proc_table[pid].wakeup_events = 0;
310016             proc_table[pid].wakeup_inode = NULL;
310017             proc_table[pid].status = PROC_READY;
310018         }
310019     }
310020 }

```

#### 94.14.25 kernel/proc/proc\_wakeup\_pipe\_write.c

« Si veda la sezione 93.20.23.

```

311001 #include <kernel/proc.h>
311002 //-----
311003 void
311004 proc_wakeup_pipe_write (inode_t * inode)
311005 {
311006     pid_t pid;
311007
311008     for (pid = 1; pid < PROCESS_MAX; pid++)
311009     {
311010         if ((proc_table[pid].status == PROC_SLEEPING)
311011             && (proc_table[pid].wakeup_events
311012                & WAKEUP_EVENT_PIPE_WRITE)
311013             && (proc_table[pid].wakeup_inode == inode))
311014         {
311015             proc_table[pid].wakeup_events = 0;
311016             proc_table[pid].wakeup_inode = NULL;
311017             proc_table[pid].status = PROC_READY;
311018         }
311019     }
311020 }

```

#### 94.14.26 kernel/proc/proc\_wakeup\_terminal.c

« Si veda la sezione 93.20.23.

```

312001 #include <kernel/proc.h>
312002 #include <kernel/lib_k.h>
312003 #include <sys/types.h>
312004 //-----
312005 void
312006 proc_wakeup_terminal (void)
312007 {
312008     pid_t pid;
312009     int maj;
312010     //
312011     // At the moment, all processes waiting for reading
312012     // a terminal
312013     // or the console are reactivated.
312014     //
312015     for (pid = 0; pid < PROCESS_MAX; pid++)
312016     {
312017         if ((proc_table[pid].status == PROC_SLEEPING)
312018             && (proc_table[pid].wakeup_events &
312019                WAKEUP_EVENT_DEV_READ))
312020         {
312021             maj = major (proc_table[pid].wakeup_dev);
312022             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 static void
3140591 sysroutine_error_back (int *number, int *line,
3140592                      char *file_name)
3140593 {
3140594     *number = errno;
3140595     *line = errln;
3140596     strncpy (file_name, errfn, PATH_MAX);
3140597     file_name[PATH_MAX - 1] = 0;
3140598 }
3140599

```



## Sorgenti della libreria generale

95.1	os32: file isolati della directory «lib/»	753
95.1.1	lib/NULL.h	753
95.1.2	lib/SEEK.h	753
95.1.3	lib/assert.h	753
95.1.4	lib/clock_t.h	754
95.1.5	lib/ctype.h	754
95.1.6	lib/limits.h	754
95.1.7	lib/ptrdiff_t.h	755
95.1.8	lib/restrict.h	755
95.1.9	lib/size_t.h	756
95.1.10	lib/stdarg.h	756
95.1.11	lib/stdbool.h	756
95.1.12	lib/stddef.h	756
95.1.13	lib/stdint.h	756
95.1.14	lib/time_t.h	758
95.1.15	lib/wchar_t.h	758
95.2	os32: «lib/_gcc.h»	758
95.2.1	lib/_gcc/_divdi3.c	759
95.2.2	lib/_gcc/_moddi3.c	759
95.2.3	lib/_gcc/_udivdi3.c	759
95.2.4	lib/_gcc/_umoddi3.c	759
95.2.5	lib/_gcc/_lldiv.c	759
95.2.6	lib/_gcc/_ulldiv.c	760
95.3	os32: «lib/arpa/inet.h»	761
95.3.1	lib/arpa/inet/htonl.c	761
95.3.2	lib/arpa/inet/htons.c	762
95.3.3	lib/arpa/inet/inet_ntop.c	762
95.3.4	lib/arpa/inet/inet_pton.c	762
95.3.5	lib/arpa/inet/ntohl.c	764
95.3.6	lib/arpa/inet/ntohs.c	764
95.4	os32: «lib/dirent.h»	764
95.4.1	lib/dirent/DIR.c	765
95.4.2	lib/dirent/closedir.c	765
95.4.3	lib/dirent/opendir.c	766
95.4.4	lib/dirent/readdir.c	767
95.4.5	lib/dirent/rewinddir.c	768
95.5	os32: «lib/errno.h»	768
95.5.1	lib/errno/errno.c	773
95.6	os32: «lib/fcntl.h»	773
95.6.1	lib/fcntl/creat.c	774
95.6.2	lib/fcntl/fcntl.c	775
95.6.3	lib/fcntl/open.c	775
95.7	os32: «lib/grp.h»	776
95.7.1	lib/grp/grent.c	776
95.8	os32: «lib/inttypes.h»	777
95.8.1	lib/inttypes/imaxabs.c	781
95.8.2	lib/inttypes/imaxdiv.c	781
95.9	os32: «lib/libgen.h»	781
95.9.1	lib/libgen/basename.c	781
95.9.2	lib/libgen/dirname.c	782
95.10	os32: «lib/netinet/icmp.h»	783
95.11	os32: «lib/netinet/in.h»	785

95.12	os32: «lib/netinet/ip.h»	786
95.13	os32: «lib/netinet/tcp.h»	787
95.14	os32: «lib/netinet/udp.h»	788
95.15	os32: «lib/pwd.h»	788
95.15.1	lib/pwd/pwent.c	789
95.16	os32: «lib/setjmp.h»	790
95.16.1	lib/setjmp/longjmp.c	791
95.16.2	lib/setjmp/setjmp.s	791
95.17	os32: «lib/signal.h»	792
95.17.1	lib/signal/_sighandler_wrapper.s	793
95.17.2	lib/signal/kill.c	794
95.17.3	lib/signal/signal.c	794
95.18	os32: «lib/stdio.h»	794
95.18.1	lib/stdio/FILE.c	796
95.18.2	lib/stdio/clearerr.c	797
95.18.3	lib/stdio/fclose.c	797
95.18.4	lib/stdio/feof.c	797
95.18.5	lib/stdio/ferror.c	797
95.18.6	lib/stdio/fflush.c	798
95.18.7	lib/stdio/fgetc.c	798
95.18.8	lib/stdio/fgetpos.c	798
95.18.9	lib/stdio/fgets.c	798
95.18.10	lib/stdio/fileno.c	799
95.18.11	lib/stdio/fopen.c	799
95.18.12	lib/stdio/fprintf.c	800
95.18.13	lib/stdio/fputc.c	800
95.18.14	lib/stdio/fputs.c	801
95.18.15	lib/stdio/fread.c	801
95.18.16	lib/stdio/freopen.c	801
95.18.17	lib/stdio/fscanf.c	802
95.18.18	lib/stdio/fseek.c	802
95.18.19	lib/stdio/fseeko.c	802
95.18.20	lib/stdio/fsetpos.c	802
95.18.21	lib/stdio/ftell.c	803
95.18.22	lib/stdio/ftello.c	803
95.18.23	lib/stdio/fwrite.c	803
95.18.24	lib/stdio/getchar.c	803
95.18.25	lib/stdio/gets.c	804
95.18.26	lib/stdio/perror.c	804
95.18.27	lib/stdio/printf.c	805
95.18.28	lib/stdio/putchar.c	805
95.18.29	lib/stdio/puts.c	805
95.18.30	lib/stdio/rewind.c	806
95.18.31	lib/stdio/scanf.c	806
95.18.32	lib/stdio/setbuf.c	806
95.18.33	lib/stdio/setvbuf.c	806
95.18.34	lib/stdio/snprintf.c	806
95.18.35	lib/stdio/sprintf.c	806
95.18.36	lib/stdio/sscanf.c	807
95.18.37	lib/stdio/vfprintf.c	807
95.18.38	lib/stdio/vfscanf.c	807
95.18.39	lib/stdio/vfscanf.c	807
95.18.40	lib/stdio/vprintf.c	827
95.18.41	lib/stdio/vscanf.c	827
95.18.42	lib/stdio/vsnprintf.c	828
95.18.43	lib/stdio/vsprintf.c	843

95.18.44	lib/stdio/vsscanf.c	844
95.19	os32: «lib/stdlib.h»	844
95.19.1	lib/stdlib/_Exit.c	846
95.19.2	lib/stdlib/abort.c	846
95.19.3	lib/stdlib/abs.c	846
95.19.4	lib/stdlib/atexit.c	847
95.19.5	lib/stdlib/atoi.c	847
95.19.6	lib/stdlib/atol.c	848
95.19.7	lib/stdlib/div.c	848
95.19.8	lib/stdlib/environment.c	848
95.19.9	lib/stdlib/exit.c	849
95.19.10	lib/stdlib/getenv.c	850
95.19.11	lib/stdlib/labs.c	851
95.19.12	lib/stdlib/ldiv.c	851
95.19.13	lib/stdlib/labs.c	851
95.19.14	lib/stdlib/lldiv.c	851
95.19.15	lib/stdlib/putenv.c	851
95.19.16	lib/stdlib/qsrt.c	853
95.19.17	lib/stdlib/rand.c	854
95.19.18	lib/stdlib/setenv.c	855
95.19.19	lib/stdlib/strtol.c	856
95.19.20	lib/stdlib/strtoul.c	859
95.19.21	lib/stdlib/unsetenv.c	859
95.19.22	lib/stdlib/_alloc/_alloc_list.c	860
95.19.23	lib/stdlib/_alloc/free.c	861
95.19.24	lib/stdlib/_alloc/malloc.c	862
95.19.25	lib/stdlib/_alloc/realloc.c	865
95.20	os32: «lib/string.h»	866
95.20.1	lib/string/memccpy.c	867
95.20.2	lib/string/memchr.c	868
95.20.3	lib/string/memcmp.c	868
95.20.4	lib/string/memcpy.c	868
95.20.5	lib/string/memmove.c	868
95.20.6	lib/string/memset.c	869
95.20.7	lib/string/streat.c	869
95.20.8	lib/string/strchr.c	869
95.20.9	lib/string/stremp.c	870
95.20.10	lib/string/strcoll.c	870
95.20.11	lib/string/strcpy.c	870
95.20.12	lib/string/strespn.c	870
95.20.13	lib/string/strdup.c	871
95.20.14	lib/string/strerror.c	871
95.20.15	lib/string/strlen.c	873
95.20.16	lib/string/strncat.c	873
95.20.17	lib/string/strncpy.c	873
95.20.18	lib/string/strncpy.c	873
95.20.19	lib/string/strpbrk.c	874
95.20.20	lib/string/strrchr.c	874
95.20.21	lib/string/strspn.c	874
95.20.22	lib/string/strstr.c	875
95.20.23	lib/string/strtok.c	875
95.20.24	lib/string/strxfrm.c	877
95.21	os32: «lib/sys/os32.h»	877
95.21.1	lib/sys/os32/input_line.c	885
95.21.2	lib/sys/os32/ipconfig.c	887
95.21.3	lib/sys/os32/mount.c	888
95.21.4	lib/sys/os32/namep.c	888

95.21.5	lib/sys/os32/routeadd.c	890
95.21.6	lib/sys/os32/routedel.c	891
95.21.7	lib/sys/os32/sys.s	891
95.21.8	lib/sys/os32/umount.c	891
95.21.9	lib/sys/os32/z_perror.c	892
95.21.10	lib/sys/os32/z_printf.c	892
95.21.11	lib/sys/os32/z_vprintf.c	892
95.22	os32: «lib/sys/sa_family_t.h»	893
95.23	os32: «lib/sys/socket.h»	893
95.23.1	lib/sys/socket/accept.c	894
95.23.2	lib/sys/socket/bind.c	895
95.23.3	lib/sys/socket/connect.c	895
95.23.4	lib/sys/socket/listen.c	896
95.23.5	lib/sys/socket/recvfrom.c	896
95.23.6	lib/sys/socket/send.c	898
95.23.7	lib/sys/socket/socket.c	899
95.24	os32: «lib/sys/socklen_t.h»	899
95.25	os32: «lib/sys/stat.h»	899
95.25.1	lib/sys/stat/chmod.c	901
95.25.2	lib/sys/stat/fchmod.c	901
95.25.3	lib/sys/stat/fstat.c	902
95.25.4	lib/sys/stat/mkdir.c	902
95.25.5	lib/sys/stat/mknod.c	903
95.25.6	lib/sys/stat/stat.c	903
95.25.7	lib/sys/stat/umask.c	903
95.26	os32: «lib/sys/types.h»	904
95.26.1	lib/sys/types/major.c	904
95.26.2	lib/sys/types/makedev.c	904
95.26.3	lib/sys/types/minor.c	904
95.27	os32: «lib/sys/wait.h»	905
95.27.1	lib/sys/wait/wait.c	905
95.28	os32: «lib/termios.h»	905
95.28.1	lib/termios/tcgetattr.c	906
95.28.2	lib/termios/tcsetattr.c	907
95.29	os32: «lib/time.h»	907
95.29.1	lib/time/asctime.c	908
95.29.2	lib/time/clock.c	909
95.29.3	lib/time/gmtime.c	909
95.29.4	lib/time/mktime.c	911
95.29.5	lib/time/stime.c	913
95.29.6	lib/time/time.c	913
95.30	os32: «lib/unistd.h»	913
95.30.1	lib/unistd/_exit.c	915
95.30.2	lib/unistd/access.c	916
95.30.3	lib/unistd/brk.c	916
95.30.4	lib/unistd/chdir.c	917
95.30.5	lib/unistd/chown.c	917
95.30.6	lib/unistd/close.c	917
95.30.7	lib/unistd/dup.c	918
95.30.8	lib/unistd/dup2.c	918
95.30.9	lib/unistd/environ.c	918
95.30.10	lib/unistd/exec1.c	918
95.30.11	lib/unistd/execl.c	919
95.30.12	lib/unistd/execlp.c	919
95.30.13	lib/unistd/execv.c	920

95.30.14	lib/unistd/execve.c	920
95.30.15	lib/unistd/execl.c	921
95.30.16	lib/unistd/fchdir.c	922
95.30.17	lib/unistd/fchown.c	922
95.30.18	lib/unistd/fork.c	922
95.30.19	lib/unistd/getcwd.c	923
95.30.20	lib/unistd/getegid.c	923
95.30.21	lib/unistd/geteuid.c	924
95.30.22	lib/unistd/getgid.c	924
95.30.23	lib/unistd/getopt.c	924
95.30.24	lib/unistd/getpgrp.c	927
95.30.25	lib/unistd/getpid.c	927
95.30.26	lib/unistd/getppid.c	928
95.30.27	lib/unistd/getuid.c	928
95.30.28	lib/unistd/isatty.c	928
95.30.29	lib/unistd/link.c	929
95.30.30	lib/unistd/lseek.c	929
95.30.31	lib/unistd/pipe.c	929
95.30.32	lib/unistd/read.c	930
95.30.33	lib/unistd/rmdir.c	931
95.30.34	lib/unistd/sbrk.c	931
95.30.35	lib/unistd/setegid.c	932
95.30.36	lib/unistd/seteuid.c	932
95.30.37	lib/unistd/setgid.c	932
95.30.38	lib/unistd/setpgrp.c	932
95.30.39	lib/unistd/setuid.c	933
95.30.40	lib/unistd/sleep.c	933
95.30.41	lib/unistd/ttyname.c	933
95.30.42	lib/unistd/unlink.c	934
95.30.43	lib/unistd/write.c	934
95.31	os32: «lib/utime.h»	935
95.31.1	lib/utime/utime.c	935

## 95.1 os32: file isolati della directory «lib/»

### 95.1.1 lib/NULL.h

Si veda la sezione 91.3.

```

3150001 #ifndef _NULL_H
3150002 #define _NULL_H 1
3150003 //-----
3150004 #define NULL ((void *) 0)
3150005 //-----
3150006 #endif

```

### 95.1.2 lib/SEEK.h

Si veda la sezione 91.3.

```

3160001 #ifndef _SEEK_H
3160002 #define _SEEK_H 1
3160003 //-----
3160004 // These values are used inside 'stdio.h' and
3160005 // 'unistd.h'.
3160006 //-----
3160007 #define SEEK_SET 0 // From the start.
3160008 #define SEEK_CUR 1 // From current
3160009 // position.
3160010 #define SEEK_END 2 // From the end.
3160011 //-----
3160012 #endif

```

### 95.1.3 lib/assert.h

Si veda la sezione 88.6.

```

3170001 #ifndef _ASSERT_H
3170002 #define _ASSERT_H 1
3170003 //-----

```

```

3170004 #include <stdio.h>
3170005 //-----
3170006 #ifdef NDEBUG
3170007 #define assert(ignore) ((void)0)
3170008 #else
3170009 #define assert(ASSERTION)          \
3170010     ((if ((ASSERTION)==0)         \
3170011         fprintf (stderr,         \
3170012             "Assertion failed: " # ASSERTION \
3170013             ", function %s, file %s, line %u.\n", \
3170014             __func__, __FILE__, __LINE__);))
3170015 #endif
3170016 //-----
3170017 #endif

```

### 95.1.4 lib/clock\_t.h

« Si veda la sezione 91.3.

```

3180001 #ifndef _CLOCK_T_H
3180002 #define _CLOCK_T_H          1
3180003 //-----
3180004 #include <stdint.h>
3180005 //-----
3180006 typedef uint64_t clock_t;
3180007 //-----
3180008 #endif

```

### 95.1.5 lib/ctype.h

« Si veda la sezione 91.3.

```

3190001 #ifndef _CTYPE_H
3190002 #define _CTYPE_H          1
3190003 //-----
3190004 #include <NULL.h>
3190005 //-----
3190006 #define isblank(C) ((int) (C == ' ' || C == '\t'))
3190007 #define isspace(C) ((int) (C == ' ' \
3190008     || C == '\f' \
3190009     || C == '\n' \
3190010     || C == '\r' \
3190011     || C == '\t' \
3190012     || C == '\v'))
3190013 #define isdigit(C) ((int) (C >= '0' && C <= '9'))
3190014 #define isxdigit(C) \
3190015     ((int) ((C >= '0' && C <= '9') \
3190016     || (C >= 'A' && C <= 'F') \
3190017     || (C >= 'a' && C <= 'f')))
3190018 #define isupper(C) ((int) (C >= 'A' && C <= 'Z'))
3190019 #define islower(C) ((int) (C >= 'a' && C <= 'z'))
3190020 #define iscntrl(C) ((int) ((C >= 0x00 && C <= 0x1F) \
3190021     || C == 0x7F))
3190022 #define isgraph(C) ((int) (C >= 0x21 && C <= 0x7E))
3190023 #define isprint(C) ((int) (C >= 0x20 && C <= 0x7E))
3190024 #define isalpha(C) (isupper(C) || islower(C))
3190025 #define isalnum(C) (isalpha(C) || isdigit(C))
3190026 #define ispunct(C) (isgraph(C) && (!isspace(C)) \
3190027     && (!isalnum(C)))
3190028 #define tolower(C) (isupper(C) ? ((C) + 0x20) : (C))
3190029 #define toupper(C) (islower(C) ? ((C) - 0x20) : (C))
3190030 #define toascii(C) (C & 0x7F)
3190031 #define _tolower(C) (isupper(C) ? ((C) + 0x20) : (C))
3190032 #define _toupper(C) (islower(C) ? ((C) - 0x20) : (C))
3190033 //-----
3190034 #endif

```

### 95.1.6 lib/limits.h

« Si veda la sezione 91.3.

```

3200001 #ifndef _LIMITS_H
3200002 #define _LIMITS_H          1
3200003 //-----
3200004 #define CHAR_UNSIGNED      0
3200005 //-----
3200006 #define CHAR_BIT           (8)
3200007 //-----
3200008 #define SCHAR_MIN         (-0x80)
3200009 #define SCHAR_MAX         (0x7F)
3200010 #define UCHAR_MAX         (0xFF)
3200011 //-----
3200012 #ifdef CHAR_UNSIGNED
3200013 #define CHAR_MIN          (0)
3200014 #define CHAR_MAX          UCHAR_MAX
3200015 #else

```

```

3200016 #define CHAR_MIN          SCHAR_MIN
3200017 #define CHAR_MAX          SCHAR_MAX
3200018 #endif
3200019 //-----
3200020 #define MB_LEN_MAX        (16)
3200021 //-----
3200022 #define SHRT_MIN          (-0x8000)
3200023 #define SHRT_MAX          (0x7FFF)
3200024 #define USHRT_MAX         (0xFFFF)
3200025 //-----
3200026 #define INT_MIN           (-0x80000000)
3200027 #define INT_MAX           (0x7FFFFFFF)
3200028 #define UINT_MAX          (0xFFFFFFFFU)
3200029 //-----
3200030 #define LONG_MIN          (-0x80000000L)
3200031 #define LONG_MAX          (0x7FFFFFFFL)
3200032 #define ULONG_MAX         (0xFFFFFFFFUL)
3200033 //-----
3200034 #define LLONG_MIN         (-0x8000000000000000LL)
3200035 #define LLONG_MAX         (0x7FFFFFFFFFFFFFFFL)
3200036 #define ULLONG_MAX        (0xFFFFFFFFFFFFFFFFULL)
3200037 #define WORD_BIT          (32)
3200038 #define LONG_BIT          (32)
3200039 #define SSIZE_MAX         (0x7FFFFFFFL)
3200040 //-----
3200041 #define ARG_MAX            8192 // Arguments+environment
3200042 // max length.
3200043 #define ATEXIT_MAX         32 // Max "at exit"
3200044 // functions.
3200045 #define FILESIZEBITS      32 // File size needs integer
3200046 // size...
3200047 #define LINK_MAX           254 // Max links per file.
3200048 #define NAME_MAX           14 // File name max
3200049 // (Minix 1 fs).
3200050 #define OPEN_MAX           128 // Max open files per
3200051 // process.
3200052 #define PATH_MAX           1024 // Path, including
3200053 // final '\0'.
3200054 #define MAX_CANON          256 // Max bytes in
3200055 // canonical tty queue.
3200056 #define MAX_INPUT          1 // Max bytes in tty
3200057 // input queue.
3200058 //-----
3200059 #define CHLD_MAX           INT_MAX // Not used.
3200060 #define HOST_NAME_MAX     INT_MAX // Not used.
3200061 #define LOGIN_NAME_MAX    INT_MAX // Not used.
3200062 #define PAGE_SIZE         INT_MAX // Not used.
3200063 #define RE_DUP_MAX        INT_MAX // Not used.
3200064 #define STREAM_MAX        INT_MAX // Not used.
3200065 #define SYMLOOP_MAX       INT_MAX // Not used.
3200066 #define TTY_NAME_MAX      INT_MAX // Not used.
3200067 #define TZNAME_MAX        INT_MAX // Not used.
3200068 #define PIPE_MAX          INT_MAX // Not used.
3200069 #define SYMLINK_MAX       INT_MAX // Not used.
3200070 //-----
3200071 #endif

```

### 95.1.7 lib/ptrdiff\_t.h

« Si veda la sezione 91.3.

```

3210001 #ifndef _PTRDIFF_T_H
3210002 #define _PTRDIFF_T_H      1
3210003 //-----
3210004 typedef int ptrdiff_t;
3210005 //-----
3210006 #endif

```

### 95.1.8 lib/restrict.h

« Si veda la sezione 91.3.

```

3220001 #ifndef _RESTRICT_H
3220002 #define _RESTRICT_H          1
3220003 //-----
3220004 // At the moment, the GCC compiler does not support
3220005 // the 'restrict' keyword.
3220006 //-----
3220007 #define restrict /**/
3220008 //-----
3220009 #endif

```

## 95.1.9 lib/size\_t.h

Si veda la sezione 91.3.

```

3230001 #ifndef _SIZE_T_H
3230002 #define _SIZE_T_H      1
3230003 //-----
3230004 // The type 'size_t' *must* be equal to an 'int'.
3230005 //-----
3230006 typedef unsigned int size_t;
3230007 //-----
3230008 #endif

```

## 95.1.10 lib/stdarg.h

Si veda la sezione 91.3.

```

3240001 #ifndef _STDARG_H
3240002 #define _STDARG_H      1
3240003 //-----
3240004 typedef unsigned char *va_list;
3240005 //-----
3240006 #define va_start(ap, last) \
3240007     ((void)((ap) = \
3240008         ((va_list) &(last)) + (sizeof(last))))
3240009 #define va_end(ap) ((void)((ap) = 0))
3240010 #define va_copy(dest, src) \
3240011     ((void)((dest) = (va_list)(src)))
3240012 #define va_arg(ap, type) \
3240013     (((ap) = (ap) + (sizeof(type))), \
3240014     *((type *) ((ap) - (sizeof(type)))))
3240015 //-----
3240016 #endif

```

## 95.1.11 lib/stdbool.h

Si veda la sezione 91.3.

```

3250001 #ifndef _STDBOOL_H
3250002 #define _STDBOOL_H      1
3250003 //-----
3250004 #define bool      _Bool
3250005 #define true      1
3250006 #define false     0
3250007 #define __bool_true_false_are_defined 1
3250008 //-----
3250009 #endif

```

## 95.1.12 lib/stddef.h

Si veda la sezione 91.3.

```

3260001 #ifndef _STDDEF_H
3260002 #define _STDDEF_H      1
3260003 //-----
3260004 #include <ptrdiff_t.h>
3260005 #include <size_t.h>
3260006 #include <wchar_t.h>
3260007 #include <NULL.h>
3260008 //-----
3260009 #define offsetof(type, member) \
3260010     ((size_t) &((type *)0)->member)
3260011 //-----
3260012 #endif

```

## 95.1.13 lib/stdint.h

Si veda la sezione 91.3.

```

3270001 #ifndef _STDINT_H
3270002 #define _STDINT_H      1
3270003 //-----
3270004 typedef signed char int8_t;
3270005 typedef short int int16_t;
3270006 typedef int int32_t;
3270007 typedef long long int int64_t;
3270008 //
3270009 typedef unsigned char uint8_t;
3270010 typedef unsigned short int uint16_t;
3270011 typedef unsigned int uint32_t;
3270012 typedef unsigned long long int uint64_t;
3270013 //
3270014 #define INT8_MIN      (-0x80)
3270015 #define INT16_MIN     (-0x8000)
3270016 #define INT32_MIN     (-0x80000000)
3270017 #define INT64_MIN     (-0x8000000000000000LL)
3270018 //

```

```

3270019 #define INT8_MAX      0x7F
3270020 #define INT16_MAX     0x7FFF
3270021 #define INT32_MAX     0x7FFFFFFF
3270022 #define INT64_MAX     0x7FFFFFFFFFFFFFFFLL
3270023 //
3270024 #define UINT8_MAX     0xFF
3270025 #define UINT16_MAX    0xFFFF
3270026 #define UINT32_MAX    0xFFFFFFFFU
3270027 #define UINT64_MAX    0xFFFFFFFFFFFFFFFFULL
3270028 //-----
3270029 typedef signed char int_least8_t;
3270030 typedef short int int_least16_t;
3270031 typedef int int_least32_t;
3270032 typedef long long int int_least64_t;
3270033 //
3270034 typedef unsigned char uint_least8_t;
3270035 typedef unsigned short int uint_least16_t;
3270036 typedef unsigned int uint_least32_t;
3270037 typedef unsigned long long int uint_least64_t;
3270038 //
3270039 #define INT_LEAST8_MIN      (-0x80)
3270040 #define INT_LEAST16_MIN     (-0x8000)
3270041 #define INT_LEAST32_MIN     (-0x80000000)
3270042 #define INT_LEAST64_MIN     (-0x8000000000000000LL)
3270043 //
3270044 #define INT_LEAST8_MAX     0x7F
3270045 #define INT_LEAST16_MAX    0x7FFF
3270046 #define INT_LEAST32_MAX    0x7FFFFFFF
3270047 #define INT_LEAST64_MAX    0x7FFFFFFFFFFFFFFFLL
3270048 //
3270049 #define UINT_LEAST8_MAX    0xFF
3270050 #define UINT_LEAST16_MAX   0xFFFF
3270051 #define UINT_LEAST32_MAX   0xFFFFFFFFU
3270052 #define UINT_LEAST64_MAX   0xFFFFFFFFFFFFFFFFULL
3270053 //-----
3270054 #define INT8_C(VAL)      VAL
3270055 #define INT16_C(VAL)     VAL
3270056 #define INT32_C(VAL)     VAL
3270057 #define INT64_C(VAL)     VAL ## LL
3270058 //
3270059 #define UINT8_C(VAL)     VAL
3270060 #define UINT16_C(VAL)    VAL
3270061 #define UINT32_C(VAL)    VAL ## U
3270062 #define UINT64_C(VAL)    VAL ## ULL
3270063 //-----
3270064 typedef signed char int_fast8_t;
3270065 typedef int int_fast16_t;
3270066 typedef int int_fast32_t;
3270067 typedef long long int int_fast64_t;
3270068 //
3270069 typedef unsigned char uint_fast8_t;
3270070 typedef unsigned int uint_fast16_t;
3270071 typedef unsigned int uint_fast32_t;
3270072 typedef unsigned long long int uint_fast64_t;
3270073 //
3270074 #define INT_FAST8_MIN      (-0x80)
3270075 #define INT_FAST16_MIN     (-0x80000000)
3270076 #define INT_FAST32_MIN     (-0x80000000)
3270077 #define INT_FAST64_MIN     (-0x8000000000000000LL)
3270078 //
3270079 #define INT_FAST8_MAX     0x7F
3270080 #define INT_FAST16_MAX    0x7FFFFFFF
3270081 #define INT_FAST32_MAX    0x7FFFFFFF
3270082 #define INT_FAST64_MAX    0x7FFFFFFFFFFFFFFFLL
3270083 //
3270084 #define UINT_FAST8_MAX    0xFF
3270085 #define UINT_FAST16_MAX   0xFFFFFFFFU
3270086 #define UINT_FAST32_MAX   0xFFFFFFFFU
3270087 #define UINT_FAST64_MAX   0xFFFFFFFFFFFFFFFFULL
3270088 //-----
3270089 typedef int intptr_t;
3270090 typedef unsigned int uintptr_t;
3270091 //
3270092 #define INTPTR_MIN      (-0x80000000)
3270093 #define INTPTR_MAX      0x7FFFFFFF
3270094 #define UINTPTR_MAX     0xFFFFFFFFU
3270095 //
3270096 typedef long long int intmax_t;
3270097 typedef unsigned long long int uintmax_t;
3270098 //
3270099 #define INTMAX_C(VAL)      VAL ## LL
3270100 #define UINTMAX_C(VAL)     VAL ## ULL
3270101 #define INTMAX_MIN      (-INTMAX_C(0x8000000000000000))
3270102 #define INTMAX_MAX      (INTMAX_C(0x7FFFFFFFFFFFFFFF))
3270103 #define UINTMAX_MAX      (UINTMAX_C(0xFFFFFFFFFFFFFFFF))
3270104 //-----
3270105 #define PTRDIFF_MIN      (-0x80000000)

```

```

3270106 #define PTRDIFF_MAX      0x7FFFFFFF
3270107 //
3270108 #define SIG_ATOMIC_MIN   (-0x80000000)
3270109 #define SIG_ATOMIC_MAX   0x7FFFFFFF
3270110 //
3270111 #define SIZE_MAX         0xFFFFFFFFU
3270112 //
3270113 #define WCHAR_MIN        0x00000000
3270114 #define WCHAR_MAX        0xFFFFFFFFU
3270115 //
3270116 #define WINT_MIN         (-0x8000000000000000LL)
3270117 #define WINT_MAX         0x7FFFFFFFFFFFFFFFL
3270118 //-----
3270119 #endif

```

## 95.1.14 lib/time\_t.h

« Si veda la sezione 91.3.

```

3280001 #ifndef _TIME_T_H
3280002 #define _TIME_T_H      1
3280003 //-----
3280004 typedef long long int time_t;
3280005 //-----
3280006 #endif

```

## 95.1.15 lib/wchar\_t.h

« Si veda la sezione 91.3.

```

3290001 #ifndef _WCHAR_T_H
3290002 #define _WCHAR_T_H    1
3290003 //-----
3290004 typedef unsigned int wchar_t;
3290005 //-----
3290006 #endif

```

## 95.2 os32: «lib/\_gcc.h»

« Si veda la sezione 88.1.

```

3300001 #ifndef __GCC_H
3300002 #define __GCC_H      1
3300003 //-----
3300004 #include <stdlib.h>
3300005 //-----
3300006 typedef struct
3300007 {
3300008     unsigned long long int quot;
3300009     unsigned long long int rem;
3300010 } ulldiv_t;
3300011 //-----
3300012 lldiv_t _lldiv (long long int dividend,
3300013                long long int divisor);
3300014 ulldiv_t _ulldiv (unsigned long long int dividend,
3300015                  unsigned long long int divisor);
3300016 //-----
3300017 unsigned long long int __udivdi3 (unsigned long long
3300018                                   int dividend,
3300019                                   unsigned long long
3300020                                   int divisor);
3300021 unsigned long long int __umoddi3 (unsigned long long
3300022                                   int dividend,
3300023                                   unsigned long long
3300024                                   int divisor);
3300025 long long int __divdi3 (long long int dividend,
3300026                        long long int divisor);
3300027 long long int __moddi3 (long long int dividend,
3300028                        long long int divisor);
3300029 //-----
3300030 #endif

```

95.2.1	lib/_gcc/_divdi3.c	759
95.2.2	lib/_gcc/_moddi3.c	759
95.2.3	lib/_gcc/_udivdi3.c	759
95.2.4	lib/_gcc/_umoddi3.c	759
95.2.5	lib/_gcc/_lldiv.c	759
95.2.6	lib/_gcc/_ulldiv.c	760

## 95.2.1 lib/\_gcc/\_divdi3.c

Si veda la sezione 88.1.

```

3310001 #include <_gcc.h>
3310002 //-----
3310003 long long int
3310004 __divdi3 (long long int dividend, long long int divisor)
3310005 {
3310006     lldiv_t result;
3310007     result = _lldiv (dividend, divisor);
3310008     return result.quot;
3310009 }

```

## 95.2.2 lib/\_gcc/\_moddi3.c

Si veda la sezione 88.1.

```

3320001 #include <_gcc.h>
3320002 //-----
3320003 long long int
3320004 __moddi3 (long long int dividend, long long int divisor)
3320005 {
3320006     lldiv_t result;
3320007     result = _lldiv (dividend, divisor);
3320008     return result.rem;
3320009 }

```

## 95.2.3 lib/\_gcc/\_udivdi3.c

Si veda la sezione 88.1.

```

3330001 #include <_gcc.h>
3330002 //-----
3330003 unsigned long long int
3330004 __udivdi3 (unsigned long long int dividend,
3330005            unsigned long long int divisor)
3330006 {
3330007     ulldiv_t result;
3330008     result = _ulldiv (dividend, divisor);
3330009     return result.quot;
3330010 }

```

## 95.2.4 lib/\_gcc/\_umoddi3.c

Si veda la sezione 88.1.

```

3340001 #include <_gcc.h>
3340002 //-----
3340003 unsigned long long int
3340004 __umoddi3 (unsigned long long int dividend,
3340005            unsigned long long int divisor)
3340006 {
3340007     ulldiv_t result;
3340008     result = _ulldiv (dividend, divisor);
3340009     return result.rem;
3340010 }

```

## 95.2.5 lib/\_gcc/\_lldiv.c

Si veda la sezione 88.1.

```

3350001 #include <_gcc.h>
3350002 //-----
3350003 // If DIVIDEND and DIVISOR have different sign,
3350004 // the QUOTIENT is negative.
3350005 //
3350006 // The REMINDER has the same sign as the DIVISOR.
3350007 //-----
3350008 lldiv_t
3350009 _lldiv (long long int dividend, long long int divisor)
3350010 {
3350011     ulldiv_t uresult;
3350012     lldiv_t result;
3350013     //
3350014     // Check for sign.
3350015     //
3350016     if (dividend >= 0 && divisor >= 0)
3350017     {
3350018         uresult = _ulldiv ((unsigned long long) dividend,
3350019                           (unsigned long long) divisor);
3350020         result.quot = uresult.quot;
3350021         result.rem = uresult.rem;
3350022     }
3350023     else if (dividend < 0 && divisor < 0)
3350024     {
3350025         uresult =
3350026             _ulldiv ((unsigned long long) -dividend,

```

```

3350027         (unsigned long long) -divisor);
3350028     result.quot = uresult.quot;
3350029     result.rem = -uresult.rem;
3350030 }
3350031 else if (dividend < 0 && divisor >= 0)
3350032 {
3350033     uresult =
3350034     _udiv ((unsigned long long) -dividend,
3350035           (unsigned long long) divisor);
3350036     result.quot = -uresult.quot;
3350037     result.rem = uresult.rem;
3350038 }
3350039 else if (dividend >= 0 && divisor < 0)
3350040 {
3350041     uresult = _udiv ((unsigned long long) dividend,
3350042                    (unsigned long long) -divisor);
3350043     result.quot = uresult.quot;
3350044     result.rem = -uresult.rem;
3350045 }
3350046 //
3350047 return (result);
3350048 }

```

## 95.2.6 lib/\_gcc/\_udiv.c

Si veda la sezione 88.1.

```

3360001 #include <_gcc.h>
3360002 //-----
3360003 // DIVIDEND = DIVISOR * QUOTIENT + REMINDER
3360004 //
3360005 // If DIVISOR == 0,
3360006 // then QUOTIENT == 0 and REMINDER == DIVIDEND
3360007 //-----
3360008 udiv_t
3360009 _udiv (unsigned long long int dividend,
3360010        unsigned long long int divisor)
3360011 {
3360012     unsigned long long int sign;
3360013     unsigned long long int mask;
3360014     udiv_t result;
3360015     int scroll;
3360016     unsigned int size; // Bits of a long long.
3360017 //
3360018 // Division of zero will return zero.
3360019 //
3360020 if (dividend == 0)
3360021 {
3360022     result.quot = 0;
3360023     result.rem = 0;
3360024     return (result);
3360025 }
3360026 //
3360027 // Division by zero will return zero and all
3360028 // remainder.
3360029 //
3360030 if (divisor == 0)
3360031 {
3360032     result.quot = 0;
3360033     result.rem = dividend;
3360034     return (result);
3360035 }
3360036 //
3360037 // Calculate how much bits does have the type 'long
3360038 // long'.
3360039 //
3360040 size = 0;
3360041 mask = ~0LL;
3360042 //
3360043 while (mask > 0)
3360044 {
3360045     size += 8;
3360046     mask >>= 8;
3360047 }
3360048 //
3360049 // Calculate the value for 'sign' that needs to have
3360050 // the most
3360051 // significant bit to one.
3360052 //
3360053 mask = ~0LL;
3360054 mask >>= 1;
3360055 sign = ~mask;
3360056 //
3360057 // Scroll divisor to the left, as long as the first
3360058 // bit is zero.
3360059 //
3360060 for (scroll = 0; scroll < size; scroll++)

```

```

3360061 {
3360062     if (divisor & sign)
3360063     {
3360064         //
3360065         // The most significant bit is one.
3360066         //
3360067         break;
3360068     }
3360069     //
3360070     // The most significant bit is zero: scroll
3360071     // left.
3360072     //
3360073     divisor <<= 1;
3360074 }
3360075 //
3360076 //
3360077 //
3360078 result.quot = 0;
3360079 result.rem = 0;
3360080 //
3360081 for (; scroll >= 0 && divisor > 0; scroll--)
3360082 {
3360083     result.quot <<= 1;
3360084     if (dividend >= divisor)
3360085     {
3360086         result.quot |= 1LL;
3360087         dividend -= divisor;
3360088     }
3360089     divisor >>= 1;
3360090 }
3360091 //
3360092 result.rem = dividend;
3360093 //
3360094 return (result);
3360095 }

```

## 95.3 os32: «lib/arpa/inet.h»

Si veda la sezione 91.3.

```

3370001 #ifndef _ARPA_INET_H
3370002 #define _ARPA_INET_H 1
3370003 //-----
3370004 #include <stdint.h>
3370005 #include <sys/socklen_t.h>
3370006 //-----
3370007 uint32_t htonl (uint32_t host32);
3370008 uint16_t htons (uint16_t host16);
3370009 uint32_t ntohl (uint32_t net32);
3370010 uint16_t ntohs (uint16_t net16);
3370011 //-----
3370012 const char *inet_ntop (int family, const void *src,
3370013                       char *dst, socklen_t size);
3370014 int inet_pton (int family, const char *src, void *dst);
3370015 //-----
3370016 #endif

```

95.3.1	lib/arpa/inet/htonl.c	761
95.3.2	lib/arpa/inet/htons.c	762
95.3.3	lib/arpa/inet/inet_ntop.c	762
95.3.4	lib/arpa/inet/inet_pton.c	762
95.3.5	lib/arpa/inet/ntohl.c	764
95.3.6	lib/arpa/inet/ntohs.c	764

### 95.3.1 lib/arpa/inet/htonl.c

Si veda la sezione 88.11.

```

3380001 #include <arpa/inet.h>
3380002 //-----
3380003 uint32_t
3380004 htonl (uint32_t host32)
3380005 {
3380006     uint8_t *orig = (void *) &host32;
3380007     union
3380008     {
3380009         uint32_t value;
3380010         uint8_t b[4];
3380011     } dest;
3380012 //
3380013 // Convert: must revert byte order.
3380014 //
3380015 dest.b[0] = orig[3];

```

```

3380016 dest.b[1] = orig[2];
3380017 dest.b[2] = orig[1];
3380018 dest.b[3] = orig[0];
3380019 //
3380020 return (dest.value);
3380021 }

```

### 95.3.2 lib/arpa/inet/htons.c

« Si veda la sezione 88.11.

```

3390001 #include <arpa/inet.h>
3390002 //-----
3390003 uint16_t
3390004 htons (uint16_t host16)
3390005 {
3390006     uint8_t *orig = (void *) &host16;
3390007     union
3390008     {
3390009         uint16_t value;
3390010         uint8_t b[2];
3390011     } dest;
3390012 //
3390013 // Convert: must revert byte order.
3390014 //
3390015 dest.b[0] = orig[1];
3390016 dest.b[1] = orig[0];
3390017 //
3390018 return (dest.value);
3390019 }

```

### 95.3.3 lib/arpa/inet/inet\_ntop.c

« Si veda la sezione 88.66.

```

3400001 #include <arpa/inet.h>
3400002 #include <stdint.h>
3400003 #include <errno.h>
3400004 #include <string.h>
3400005 #include <stdlib.h>
3400006 //-----
3400007 const char *
3400008 inet_ntop (int family, const void *src, char *dst,
3400009           socklen_t size)
3400010 {
3400011 //
3400012 // Check family type: only IPv4 is available here.
3400013 //
3400014 if (family != AF_INET)
3400015 {
3400016     errset (EAFNOSUPPORT);
3400017     return (NULL);
3400018 }
3400019 //
3400020 // Check for NULL pointers.
3400021 //
3400022 if (src == NULL || dst == NULL)
3400023 {
3400024     errset (EINVAL);
3400025     return (NULL);
3400026 }
3400027 //
3400028 snprintf (dst, (size_t) size, "%i.%i.%i.%i",
3400029          *((in_addr_t *) src) >> 0 & 0x000000FF,
3400030          *((in_addr_t *) src) >> 8 & 0x000000FF,
3400031          *((in_addr_t *) src) >> 16 & 0x000000FF,
3400032          *((in_addr_t *) src) >> 24 & 0x000000FF);
3400033 //
3400034 // Return ok.
3400035 //
3400036 return (dst);
3400037 }

```

### 95.3.4 lib/arpa/inet/inet\_pton.c

« Si veda la sezione 88.67.

```

3410001 #include <arpa/inet.h>
3410002 #include <stdint.h>
3410003 #include <errno.h>
3410004 #include <string.h>
3410005 #include <stdlib.h>
3410006 //-----
3410007 #define INET_PTON_MAX_STRING_SIZE 31
3410008 //-----
3410009 int
3410010 inet_pton (int family, const char *src, void *dst)

```

```

3410011 {
3410012     char *t;
3410013     int ipv4[4];
3410014     int i;
3410015     in_addr_t result;
3410016     char source[INET_PTON_MAX_STRING_SIZE + 1];
3410017 //
3410018 // Check family type: only IPv4 is available here.
3410019 //
3410020 if (family != AF_INET)
3410021 {
3410022     errset (EAFNOSUPPORT);
3410023     return (-1);
3410024 }
3410025 //
3410026 // Check for NULL pointers.
3410027 //
3410028 if (src == NULL || dst == NULL)
3410029 {
3410030     errset (EINVAL);
3410031     return (-1);
3410032 }
3410033 //
3410034 // Check the source string size.
3410035 //
3410036 if (strlen (src) > INET_PTON_MAX_STRING_SIZE)
3410037 {
3410038 //
3410039 // The IPv4 address scan is finished
3410040 // prematurely:
3410041 // return zero to tell that the address string
3410042 // is
3410043 // not correct.
3410044 //
3410045     return (0);
3410046 }
3410047 //
3410048 // Copy the source address, to be able to modify
3410049 // the string.
3410050 //
3410051 strcpy (source, src);
3410052 //
3410053 // Start 'tokenize' the string: it is here
3410054 // accepted also
3410055 // the space as a delimiter.
3410056 //
3410057 t = strtok (source, ". ");
3410058 //
3410059 for (i = 0; i < 4 && t != NULL; i++)
3410060 {
3410061     ipv4[i] = atoi (t);
3410062     if (ipv4[i] > 255 || ipv4[i] < 0)
3410063     {
3410064 //
3410065 // An octet cannot have a value greater than
3410066 // 255,
3410067 // and cannot be negative.
3410068 //
3410069         break;
3410070     }
3410071     t = strtok (NULL, ". ");
3410072 }
3410073 //
3410074 if (i < 4)
3410075 {
3410076 //
3410077 // The IPv4 address scan is finished
3410078 // prematurely:
3410079 // return zero to tell that the address string
3410080 // is
3410081 // not correct.
3410082 //
3410083     return (0);
3410084 }
3410085 //
3410086 // Translate into a network byte order IPv4 address:
3410087 // the architecture is little-endian.
3410088 //
3410089 result = 0;
3410090 result += (ipv4[0] << 0) & 0x000000FF;
3410091 result += (ipv4[1] << 8) & 0x0000FF00;
3410092 result += (ipv4[2] << 16) & 0x00FF0000;
3410093 result += (ipv4[3] << 24) & 0xFF000000;
3410094 //
3410095 // Update the destination.
3410096 //
3410097 *((in_addr_t *) dst) = result;

```



```

3440098 //
3440099 // Return ok.
3440100 //
3440101 return (1);
3440102 }

```

### 95.3.5 lib/arpa/inet/ntohl.c

« Si veda la sezione 88.11.

```

3420001 #include <arpa/inet.h>
3420002 //-----
3420003 uint32_t
3420004 ntohl (uint32_t net32)
3420005 {
3420006     uint8_t *orig = (void *) &net32;
3420007     union
3420008     {
3420009         uint32_t value;
3420010         uint8_t b[4];
3420011     } dest;
3420012     //
3420013     // Convert: must revert byte order.
3420014     //
3420015     dest.b[0] = orig[3];
3420016     dest.b[1] = orig[2];
3420017     dest.b[2] = orig[1];
3420018     dest.b[3] = orig[0];
3420019     //
3420020     return (dest.value);
3420021 }

```

### 95.3.6 lib/arpa/inet/ntohs.c

« Si veda la sezione 88.11.

```

3430001 #include <arpa/inet.h>
3430002 //-----
3430003 uint16_t
3430004 ntohs (uint16_t net16)
3430005 {
3430006     uint8_t *orig = (void *) &net16;
3430007     union
3430008     {
3430009         uint16_t value;
3430010         uint8_t b[2];
3430011     } dest;
3430012     //
3430013     // Convert: must revert byte order.
3430014     //
3430015     dest.b[0] = orig[1];
3430016     dest.b[1] = orig[0];
3430017     //
3430018     return (dest.value);
3430019 }

```

### 95.4 os32: «lib/dirent.h»

« Si veda la sezione 91.3.

```

3440001 #ifndef _DIRENT_H
3440002 #define _DIRENT_H      1
3440003
3440004 #include <sys/types.h> // ino_t
3440005 #include <limits.h>   // NAME_MAX
3440006
3440007 //-----
3440008 struct dirent
3440009 {
3440010     ino_t d_ino; // I-node number [1]
3440011     char d_name[NAME_MAX + 1]; // NAME_MAX + Null
3440012     // termination
3440013     __attribute__((packed));
3440014     //
3440015     // [1] The type 'ino_t' must be equal to 'uint16_t',
3440016     // because the directory inside the Minix 1 file
3440017     // system has exactly such size.
3440018     //
3440019     //-----
3440020 #define DOPEN_MAX OPEN_MAX/2 // <limits.h> [1]
3440021 //
3440022 // [1] DOPEN_MAX is not standard, but it is used to
3440023 // define how many directory slot to keep for open
3440024 // directories. As directory streams are opened as
3440025 // file descriptors, the sum of all kind of file
3440026 // open cannot be more than OPEN_MAX.
3440027 //-----

```

```

3440028 typedef struct
3440029 {
3440030     int fdn; // File descriptor number.
3440031     struct dirent dir; // Last directory item read.
3440032 } DIR;
3440033
3440034 extern DIR _directory_stream[]; // Defined inside
3440035 // 'lib/dirent/DIR.c'.
3440036 //-----
3440037 // Function prototypes.
3440038 //-----
3440039 int closedir (DIR * dp);
3440040 DIR *opendir (const char *name);
3440041 struct dirent *readdir (DIR * dp);
3440042 void rewinddir (DIR * dp);
3440043 //-----
3440044 #endif

```

95.4.1 lib/dirent/DIR.c ..... 765

95.4.2 lib/dirent/closedir.c ..... 765

95.4.3 lib/dirent/opendir.c ..... 766

95.4.4 lib/dirent/readdir.c ..... 767

95.4.5 lib/dirent/rewinddir.c ..... 768

### 95.4.1 lib/dirent/DIR.c

« Si veda la sezione 91.3.

```

3450001 #include <dirent.h>
3450002 //
3450003 // There must be room for at least 'DOPEN_MAX'
3450004 // elements.
3450005 //
3450006 DIR _directory_stream[DOPEN_MAX];
3450007
3450008 void
3450009 _dirent_directory_stream_setup (void)
3450010 {
3450011     int d;
3450012     //
3450013     for (d = 0; d < DOPEN_MAX; d++)
3450014     {
3450015         _directory_stream[d].fdn = -1;
3450016     }
3450017 }

```

### 95.4.2 lib/dirent/closedir.c

« Si veda la sezione 88.13.

```

3460001 #include <dirent.h>
3460002 #include <fcntl.h>
3460003 #include <sys/types.h>
3460004 #include <sys/stat.h>
3460005 #include <unistd.h>
3460006 #include <errno.h>
3460007 #include <stddef.h>
3460008 //-----
3460009 int
3460010 closedir (DIR * dp)
3460011 {
3460012     //
3460013     // Check for a valid argument
3460014     //
3460015     if (dp == NULL)
3460016     {
3460017         //
3460018         // Not a valid pointer.
3460019         //
3460020         errset (EBADF); // Invalid directory.
3460021         return (-1);
3460022     }
3460023     //
3460024     // Check if it is an open directory stream.
3460025     //
3460026     if (dp->fdn < 0)
3460027     {
3460028         //
3460029         // The stream is closed.
3460030         //
3460031         errset (EBADF); // Invalid directory.
3460032         return (-1);
3460033     }

```

```

3460034 //
3460035 // Close the file descriptor. If there is an error,
3460036 // the 'errno' variable will be set by 'close()'.
3460037 //
3460038 return (close (dp->fdn));
3460039 }

```

### 95.4.3 lib/dirent/ opendir.c

« Si veda la sezione 88.89.

```

3470001 #include <dirent.h>
3470002 #include <fcntl.h>
3470003 #include <stdio.h>
3470004 #include <sys/types.h>
3470005 #include <sys/stat.h>
3470006 #include <unistd.h>
3470007 #include <errno.h>
3470008 #include <stddef.h>
3470009 //-----
3470010 DIR *
3470011 opendir (const char *path)
3470012 {
3470013     int fdn;
3470014     int d;
3470015     DIR *dp;
3470016     struct stat file_status;
3470017 //
3470018 // Function 'opendir()' is used only for reading.
3470019 //
3470020 fdn = open (path, O_RDONLY);
3470021 //
3470022 // Check the file descriptor returned.
3470023 //
3470024 if (fdn < 0)
3470025 {
3470026 //
3470027 // The variable 'errno' is already set:
3470028 // EINVAL
3470029 // EMFILE
3470030 // ENFILE
3470031 //
3470032 errset (errno);
3470033 return (NULL);
3470034 }
3470035 //
3470036 // Set the 'FD_CLOEXEC' flag for that file
3470037 // descriptor.
3470038 //
3470039 if (fcntl (fdn, F_SETFD, FD_CLOEXEC) != 0)
3470040 {
3470041 //
3470042 // The variable 'errno' is already set:
3470043 // EBADF
3470044 //
3470045 errset (errno);
3470046 close (fdn);
3470047 return (NULL);
3470048 }
3470049 //
3470050 //
3470051 //
3470052 if (fstat (fdn, &file_status) != 0)
3470053 {
3470054 //
3470055 // Error should be already set.
3470056 //
3470057 errset (errno);
3470058 close (fdn);
3470059 return (NULL);
3470060 }
3470061 //
3470062 // Verify it is a directory
3470063 //
3470064 if (!S_ISDIR (file_status.st_mode))
3470065 {
3470066 //
3470067 // It is not a directory!
3470068 //
3470069 close (fdn);
3470070 errset (ENOTDIR); // Is not a directory.
3470071 return (NULL);
3470072 }
3470073 //
3470074 // A valid file descriptor is available: must find a
3470075 // free
3470076 // '_directory_stream[]' slot.

```

```

3470077 //
3470078 for (d = 0; d < DOPEN_MAX; d++)
3470079 {
3470080     if (_directory_stream[d].fdn < 0)
3470081     {
3470082 //
3470083 // Found a free slot: set it up.
3470084 //
3470085 dp = &(_directory_stream[d]);
3470086 dp->fdn = fdn;
3470087 //
3470088 // Return the directory pointer.
3470089 //
3470090 return (dp);
3470091 }
3470092 }
3470093 //
3470094 // If we are here, there was no free directory slot
3470095 // available.
3470096 //
3470097 close (fdn);
3470098 errset (EMFILE); // Too many file open.
3470099 return (NULL);
3470100 }

```

### 95.4.4 lib/dirent/ readdir.c

« Si veda la sezione 88.98.

```

3480001 #include <dirent.h>
3480002 #include <fcntl.h>
3480003 #include <sys/types.h>
3480004 #include <sys/stat.h>
3480005 #include <unistd.h>
3480006 #include <errno.h>
3480007 #include <stddef.h>
3480008 //-----
3480009 struct dirent *
3480010 readdir (DIR * dp)
3480011 {
3480012     ssize_t size;
3480013 //
3480014 // Check for a valid argument.
3480015 //
3480016 if (dp == NULL)
3480017 {
3480018 //
3480019 // Not a valid pointer.
3480020 //
3480021 errset (EBADF); // Invalid directory.
3480022 return (NULL);
3480023 }
3480024 //
3480025 // Check if it is an open directory stream.
3480026 //
3480027 if (dp->fdn < 0)
3480028 {
3480029 //
3480030 // The stream is closed.
3480031 //
3480032 errset (EBADF); // Invalid directory.
3480033 return (NULL);
3480034 }
3480035 //
3480036 // Read the directory.
3480037 //
3480038 size = read (dp->fdn, &(dp->dir), (size_t) 16);
3480039 //
3480040 // Fix the null termination, if the name is very
3480041 // long.
3480042 //
3480043 dp->dir.d_name[NAME_MAX] = '\0';
3480044 //
3480045 // Check what was read.
3480046 //
3480047 if (size == 0)
3480048 {
3480049 //
3480050 // End of directory, but it is not an error.
3480051 //
3480052 return (NULL);
3480053 }
3480054 //
3480055 if (size < 0)
3480056 {
3480057 //
3480058 // This is an error. The variable 'errno' is

```

```

3480059 // already set.
3480060 //
3480061 errset (errno);
3480062 return (NULL);
3480063 }
3480064 //
3480065 if (dp->dir.d_ino == 0)
3480066 {
3480067 //
3480068 // This is a null directory record.
3480069 // Should try to read the next one.
3480070 //
3480071 return (readdir (dp));
3480072 }
3480073 //
3480074 if (strlen (dp->dir.d_name) == 0)
3480075 {
3480076 //
3480077 // This is a bad directory record: try to read
3480078 // next.
3480079 //
3480080 return (readdir (dp));
3480081 }
3480082 //
3480083 // A valid directory record should be available now.
3480084 //
3480085 return (&(dp->dir));
3480086 }

```

### 95.4.5 lib/dirent/rewinddir.c

«

Si veda la sezione 88.101.

```

3490001 #include <dirent.h>
3490002 #include <fcntl.h>
3490003 #include <sys/types.h>
3490004 #include <sys/stat.h>
3490005 #include <unistd.h>
3490006 #include <errno.h>
3490007 #include <stddef.h>
3490008 #include <stdio.h>
3490009 //-----
3490010 void
3490011 rewinddir (DIR * dp)
3490012 {
3490013 FILE *fp;
3490014 //
3490015 // Check for a valid argument.
3490016 //
3490017 if (dp == NULL)
3490018 {
3490019 //
3490020 // Nothing to rewind, and no error to set.
3490021 //
3490022 return;
3490023 }
3490024 //
3490025 // Check if it is an open directory stream.
3490026 //
3490027 if (dp->fdn < 0)
3490028 {
3490029 //
3490030 // The stream is closed.
3490031 // Nothing to rewind, and no error to set.
3490032 //
3490033 return;
3490034 }
3490035 //
3490036 //
3490037 //
3490038 fp = &_stream[dp->fdn];
3490039 //
3490040 rewind (fp);
3490041 }

```

### 95.5 os32: «lib/errno.h»

«

Si veda la sezione 88.20.

```

3500001 #ifndef _ERRNO_H
3500002 #define _ERRNO_H 1
3500003 //-----
3500004 #include <limits.h>
3500005 #include <string.h>
3500006 #include <sys/os32.h>
3500007 #include <kernel/lib_k.h>
3500008

```

```

3500009 //-----
3500010 // The variable 'errno' is standard, but 'errln' and
3500011 // 'errfn' are added to keep track of the error source.
3500012 // Variable 'errln' is used to save the source file
3500013 // line number; variable 'errfn' is used to save the
3500014 // source file name. To set these variable in a
3500015 // consistent way it is also added a macroinstruction:
3500016 // 'errset'.
3500017 //-----
3500018 extern int errno;
3500019 extern int errln;
3500020 extern char errfn[PATH_MAX];
3500021 //
3500022 #define errset(e) \
3500023 (errln = __LINE__, \
3500024 strncpy (errfn, __FILE__, PATH_MAX), \
3500025 errno = e)
3500026 //-----
3500027 // Standard POSIX 'errno' macro variables.
3500028 //-----
3500029 #define E2BIG 1 // Argument list too
3500030 // long.
3500031 #define EACCES 2 // Permission denied.
3500032 #define EADDRINUSE 3 // Address in use.
3500033 #define EADDRNOTAVAIL 4 // Address not
3500034 // available.
3500035 #define EAFNOSUPPORT 5 // Address family not
3500036 // supported.
3500037 #define EAGAIN 6 // Resource
3500038 // unavailable, try
3500039 // again.
3500040 #define EALREADY 7 // Connection already
3500041 // in progress.
3500042 #define EBADF 8 // Bad file
3500043 // descriptor.
3500044 #define EBADMSG 9 // Bad message.
3500045 #define EBUSY 10 // Device or resource
3500046 // busy.
3500047 #define ECANCELED 11 // Operation canceled.
3500048 #define ECHILD 12 // No child processes.
3500049 #define ECONNABORTED 13 // Connection aborted.
3500050 #define ECONNREFUSED 14 // Connection refused.
3500051 #define ECONNRESET 15 // Connection reset.
3500052 #define EDEADLK 16 // Resource deadlock
3500053 // would occur.
3500054 #define EDESTADDRREQ 17 // Destination address
3500055 // required.
3500056 #define EDOM 18 // Mathematics
3500057 // argument out of
3500058 // domain of
3500059 // function.
3500060 #define EDQUOT 19 // Reserved.
3500061 #define EEXIST 20 // File exists.
3500062 #define EFAULT 21 // Bad address.
3500063 #define EFBIG 22 // File too large.
3500064 #define EHOSTUNREACH 23 // Host is
3500065 // unreachable.
3500066 #define EIDRM 24 // Identifier removed.
3500067 #define EILSEQ 25 // Illegal byte
3500068 // sequence.
3500069 #define EINPROGRESS 26 // Operation in
3500070 // progress.
3500071 #define EINTR 27 // Interrupted
3500072 // function.
3500073 #define EINVAL 28 // Invalid argument.
3500074 #define EIO 29 // I/O error.
3500075 #define EISCONN 30 // Socket is
3500076 // connected.
3500077 #define EISDIR 31 // Is a directory.
3500078 #define ELOOP 32 // Too many levels of
3500079 // symbolic links.
3500080 #define EMFILE 33 // Too many open
3500081 // files.
3500082 #define EMLINK 34 // Too many links.
3500083 #define EMSGSIZE 35 // Message too large.
3500084 #define EMULTIHOP 36 // Reserved.
3500085 #define ENAMETOOLONG 37 // Filename too long.
3500086 #define ENETDOWN 38 // Network is down.
3500087 #define ENETRESET 39 // Connection aborted
3500088 // by network.
3500089 #define ENETUNREACH 40 // Network
3500090 // unreachable.
3500091 #define ENFILE 41 // Too many files open
3500092 // in system.
3500093 #define ENOBUFS 42 // No buffer space
3500094 // available.
3500095 #define ENODATA 43 // No message is

```

```

350096 // available on the
350097 // stream head
350098 // read queue.
350099 #define ENODEV 44 // No such device.
350100 #define ENOENT 45 // No such file or
350101 // directory.
350102 #define ENOEXEC 46 // Executable file
350103 // format error.
350104 #define ENOLCK 47 // No locks available.
350105 #define ENOLINK 48 // Reserved.
350106 #define ENOMEM 49 // Not enough space.
350107 #define ENOMSG 50 // No message of the
350108 // desired type.
350109 #define ENOPROTOPT 51 // Protocol not
350110 // available.
350111 #define ENOSPC 52 // No space left on
350112 // device.
350113 #define ENOSR 53 // No stream
350114 // resources.
350115 #define ENOSTR 54 // Not a stream.
350116 #define ENOSYS 55 // Function not
350117 // supported.
350118 #define ENOTCONN 56 // The socket is not
350119 // connected.
350120 #define ENOTDIR 57 // Not a directory.
350121 #define ENOTEMPTY 58 // Directory not
350122 // empty.
350123 #define ENOTSOCK 59 // Not a socket.
350124 #define ENOTSUP 60 // Not supported.
350125 #define ENOTTY 61 // Inappropriate I/O
350126 // control operation.
350127 #define ENXIO 62 // No such device or
350128 // address.
350129 #define EOPNOTSUPP 63 // Operation not
350130 // supported on
350131 // socket.
350132 #define EOVERFLOW 64 // Value too large to
350133 // be stored in data
350134 // type.
350135 #define EPERM 65 // Operation not
350136 // permitted.
350137 #define EPIPE 66 // Broken pipe.
350138 #define EPROTO 67 // Protocol error.
350139 #define EPROTONOSUPPORT 68 // Protocol not
350140 // supported.
350141 #define EPROTOTYPE 69 // Protocol wrong type
350142 // for socket.
350143 #define ERANGE 70 // Result too large.
350144 #define EROFS 71 // Read-only file
350145 // system.
350146 #define ESPIPE 72 // Invalid seek.
350147 #define ESRCH 73 // No such process.
350148 #define ESTALE 74 // Reserved.
350149 #define ETIME 75 // Stream ioctl()
350150 // timeout.
350151 #define ETIMEDOUT 76 // Connection timed
350152 // out.
350153 #define ETXTBSY 77 // Text file busy.
350154 #define EWOULDBLOCK 78 // Operation would
350155 // block (may be the
350156 // same as EAGAIN).
350157 #define EXDEV 79 // Cross-device link.
350158 //-----
350159 // Added os32 errors.
350160 //-----
350161 #define EUNKNOWN (-1) // Unknown
350162 // error.
350163 #define E_NO_MEDIUM 80 // No medium
350164 // found.
350165 #define E_MEDIUM 81 // Medium
350166 // reported
350167 // error.
350168 #define E_FILE_TYPE 82 // File type
350169 // not
350170 // compatible.
350171 #define E_ROOT_INODE_NOT_CACHED 83 // The root
350172 // directory
350173 // inode is
350174 // not cached.
350175 #define E_CANNOT_READ_SUPERBLOCK 84 // Cannot read
350176 // super
350177 // block.
350178 #define E_MAP_INODE_TOO_BIG 85 // Map inode
350179 // too big.
350180 #define E_MAP_ZONE_TOO_BIG 86 // Map zone
350181 // too big.
350182 #define E_DATA_ZONE_TOO_BIG 87 // Data zone

```

```

350183 // too big.
350184 #define E_CANNOT_FIND_ROOT_DEVICE 88 // Cannot find
350185 // root
350186 // device.
350187 #define E_CANNOT_FIND_ROOT_INODE 89 // Cannot find
350188 // root inode.
350189 #define E_FILE_TYPE_UNSUPPORTED 90 // File type
350190 // unsupported.
350191 #define E_ENV_TOO_BIG 91 // Environment
350192 // too big.
350193 #define E_LIMIT 92 // Exceeded
350194 // implementa-
350195 // tion limits.
350196 #define E_NOT_MOUNTED 93 // Not
350197 // mounted.
350198 #define E_NOT_IMPLEMENTED 94 // Not
350199 // implemented.
350200 #define E_HARDWARE_FAULT 95 // Hardware
350201 // fault.
350202 #define E_DRIVER_FAULT 96 // Driver
350203 // fault.
350204 #define E_PIPE_FULL 97 // Pipe full.
350205 #define E_PIPE_EMPTY 98 // Pipe empty.
350206 #define E_PART_TYPE_NOT_MINIX 99 // Not a Minix
350207 // partition
350208 // type.
350209 #define E_FS_TYPE_NOT_SUPPORTED 100 // File system
350210 // type not
350211 // supported.
350212 #define E_PDU_TOO_BIG 101 // PDU too
350213 // big.
350214 #define E_ARP_MISSING 102 // ARP missing
350215 // address.
350216 //-----
350217 // Default descriptions for errors.
350218 //-----
350219 #define TEXT_E2BIG "Argument list too long."
350220 #define TEXT_EACCES "Permission denied."
350221 #define TEXT_EADDRINUSE "Address in use."
350222 #define TEXT_EADDRNOTAVAIL "Address not available."
350223 #define TEXT_EAFNOSUPPORT "Address family not " \
350224 "supported."
350225 #define TEXT_EAGAIN "Resource unavailable, " \
350226 "try again."
350227 #define TEXT_EALREADY "Connection already in " \
350228 "progress."
350229 #define TEXT_EBADF "Bad file descriptor."
350230 #define TEXT_EBADMSG "Bad message."
350231 #define TEXT_EBUSY "Device or resource busy."
350232 #define TEXT_ECANCELED "Operation canceled."
350233 #define TEXT_ECHILD "No child processes."
350234 #define TEXT_ECONNABORTED "Connection aborted."
350235 #define TEXT_ECONNREFUSED "Connection refused."
350236 #define TEXT_ECONNRESET "Connection reset."
350237 #define TEXT_EDEADLK "Resource deadlock " \
350238 "would occur."
350239 #define TEXT_EDESTADDRREQ "Destination address " \
350240 "required."
350241 #define TEXT_EDOM "Mathematics argument " \
350242 "out of " \
350243 "domain of function."
350244 #define TEXT_EDQUOT "Reserved error: EDQUOT"
350245 #define TEXT_EEXIST "File exists."
350246 #define TEXT_EFAULT "Bad address."
350247 #define TEXT_EFBIG "File too large."
350248 #define TEXT_EHOSTUNREACH "Host is unreachable."
350249 #define TEXT_EIDRM "Identifier removed."
350250 #define TEXT_EILSEQ "Illegal byte sequence."
350251 #define TEXT_EINPROGRESS "Operation in progress."
350252 #define TEXT_EINTR "Interrupted function."
350253 #define TEXT_EINVAL "Invalid argument."
350254 #define TEXT_EIO "I/O error."
350255 #define TEXT_EISCONN "Socket is connected."
350256 #define TEXT_EISDIR "Is a directory."
350257 #define TEXT_ELOOP "Too many levels of " \
350258 "symbolic links."
350259 #define TEXT_EMFILE "Too many open files."
350260 #define TEXT_EMLINK "Too many links."
350261 #define TEXT_EMSGSIZE "Message too large."
350262 #define TEXT_EMULTIHOP "Reserved error: " \
350263 "EMULTIHOP"
350264 #define TEXT_ENAMETOOLONG "Filename too long."
350265 #define TEXT_ENETDOWN "Network is down."
350266 #define TEXT_ENETRESET "Connection aborted by " \
350267 "network."
350268 #define TEXT_ENETUNREACH "Network unreachable."
350269 #define TEXT_ENFILE "Too many files open " \

```

```

3500270 "in system."
3500271 #define TEXT_ENOBUFFS "No buffer space " \
3500272 "available."
3500273 #define TEXT_ENODATA "No message is " \
3500274 "available on the " \
3500275 "stream head read queue."
3500276 #define TEXT_ENODEV "No such device."
3500277 #define TEXT_ENOENT "No such file or " \
3500278 "directory."
3500279 #define TEXT_ENOEXEC "Executable file " \
3500280 "format error."
3500281 #define TEXT_ENOLCK "No locks available."
3500282 #define TEXT_ENOLINK "Reserved error: ENOLINK"
3500283 #define TEXT_ENOMEM "Not enough space."
3500284 #define TEXT_ENOMSG "No message of the " \
3500285 "desired type."
3500286 #define TEXT_ENOPROTOOPT "Protocol not available."
3500287 #define TEXT_ENOSPC "No space left on device."
3500288 #define TEXT_ENOSR "No stream resources."
3500289 #define TEXT_ENOSTR "Not a stream."
3500290 #define TEXT_ENOSYS "Function not supported."
3500291 #define TEXT_ENOTCONN "The socket is not " \
3500292 "connected."
3500293 #define TEXT_ENOTDIR "Not a directory."
3500294 #define TEXT_ENOTEMPTY "Directory not empty."
3500295 #define TEXT_ENOTSOCK "Not a socket."
3500296 #define TEXT_ENOTSUP "Not supported."
3500297 #define TEXT_ENOTTY "Inappropriate I/O " \
3500298 "control operation."
3500299 #define TEXT_ENXIO "No such device or " \
3500300 "address."
3500301 #define TEXT_EOPNOTSUPP "Operation not " \
3500302 "supported on socket."
3500303 #define TEXT_EOVERFLOW "Value too large to be " \
3500304 "stored in data type."
3500305 #define TEXT_EPERM "Operation not permitted."
3500306 #define TEXT_EPIPE "Broken pipe."
3500307 #define TEXT_EPROTO "Protocol error."
3500308 #define TEXT_EPROTONOSUPPORT "Protocol not supported."
3500309 #define TEXT_EPROTOTYPE "Protocol wrong type " \
3500310 "for socket."
3500311 #define TEXT_ERANGE "Result too large."
3500312 #define TEXT_EROFS "Read-only file system."
3500313 #define TEXT_ESPIPE "Invalid seek."
3500314 #define TEXT_ESRCH "No such process."
3500315 #define TEXT_ESTALE "Reserved error: ESTALE"
3500316 #define TEXT_ETIME "Stream ioctl() timeout."
3500317 #define TEXT_ETIMEDOUT "Connection timed out."
3500318 #define TEXT_ETXTBSY "Text file busy."
3500319 #define TEXT_EWOULDBLOCK "Operation would block."
3500320 #define TEXT_EXDEV "Cross-device link."
3500321 //-----
3500322 #define TEXT_EUNKNOWN \
3500323 "Unknown error."
3500324 #define TEXT_E_NO_MEDIUM \
3500325 "No medium found."
3500326 #define TEXT_E_MEDIUM \
3500327 "Medium reported error"
3500328 #define TEXT_E_FILE_TYPE \
3500329 "File type not compatible."
3500330 #define TEXT_E_ROOT_INODE_NOT_CACHED \
3500331 "The root directory inode is not cached."
3500332 #define TEXT_E_CANNOT_READ_SUPERBLOCK \
3500333 "Cannot read super block."
3500334 #define TEXT_E_MAP_INODE_TOO_BIG \
3500335 "Map inode too big."
3500336 #define TEXT_E_MAP_ZONE_TOO_BIG \
3500337 "Map zone too big."
3500338 #define TEXT_E_DATA_ZONE_TOO_BIG \
3500339 "Data zone too big."
3500340 #define TEXT_E_CANNOT_FIND_ROOT_DEVICE \
3500341 "Cannot find root device."
3500342 #define TEXT_E_CANNOT_FIND_ROOT_INODE \
3500343 "Cannot find root inode."
3500344 #define TEXT_E_FILE_TYPE_UNSUPPORTED \
3500345 "File type unsupported."
3500346 #define TEXT_E_ENV_TOO_BIG \
3500347 "Environment too big."
3500348 #define TEXT_E_LIMIT \
3500349 "Exceeded implementation limits."
3500350 #define TEXT_E_NOT_MOUNTED \
3500351 "Not mounted."
3500352 #define TEXT_E_NOT_IMPLEMENTED \
3500353 "Not implemented."
3500354 #define TEXT_E_HARDWARE_FAULT \
3500355 "Hardware fault."
3500356 #define TEXT_E_DRIVER_FAULT \

```

```

3500357 "Driver fault."
3500358 #define TEXT_E_PIPE_FULL \
3500359 "Pipe full."
3500360 #define TEXT_E_PIPE_EMPTY \
3500361 "Pipe empty."
3500362 #define TEXT_E_PART_TYPE_NOT_MINIX \
3500363 "Not a Minix partition type."
3500364 #define TEXT_E_FS_TYPE_NOT_SUPPORTED \
3500365 "File system type not supported."
3500366 #define TEXT_E_FDU_TOO_BIG \
3500367 "PDU too big."
3500368 #define TEXT_E_ARP_MISSING \
3500369 "ARP missing address."
3500370 //-----
3500371 #endif

```

95.5.1 lib/errno/errno.c ..... 773

95.5.1 lib/errno/errno.c

Si veda la sezione 88.20.

```

3510001 //-----
3510002 // This file does not include the 'errno.h' header,
3510003 // because here 'errno' should not be declared as an
3510004 // extern variable!
3510005 //-----
3510006 #include <limits.h>
3510007 //-----
3510008 // The variable 'errno' is standard, but 'errln' and
3510009 // 'errfn' are added to keep track of the error source.
3510010 // Variable 'errln' is used to save the source file
3510011 // line number; variable 'errfn' is used to save the
3510012 // source file name.
3510013 // To set these variable in a consistent way it is
3510014 // also added a macroinstruction: 'errset'.
3510015 //-----
3510016 int errno;
3510017 int errln;
3510018 char errfn[PATH_MAX];
3510019 //-----

```

95.6 os32: «lib/fcntl.h»

Si veda la sezione 91.3.

```

3520001 #ifndef _FCNTL_H
3520002 #define _FCNTL_H 1
3520003
3520004 #include <sys/types.h> // mode_t
3520005 // off_t
3520006 // pid_t
3520007 //-----
3520008 // Values for the second parameter of function
3520009 // 'fcntl()'.
3520010 //-----
3520011 #define F_DUPFD 0 // Duplicate file
3520012 // descriptor.
3520013 #define F_GETFD 1 // Get file descriptor
3520014 // flags.
3520015 #define F_SETFD 2 // Set file descriptor
3520016 // flags.
3520017 #define F_GETFL 3 // Get file status
3520018 // flags.
3520019 #define F_SETFL 4 // Set file status
3520020 // flags.
3520021 #define F_GETLK 5 // Get record locking
3520022 // information.
3520023 #define F_SETLK 6 // Set record locking
3520024 // information.
3520025 #define F_SETLKW 7 // Set record locking
3520026 // information;
3520027 // wait if blocked.
3520028 #define F_GETOWN 8 // Set owner of
3520029 // socket.
3520030 #define F_SETOWN 9 // Get owner of
3520031 // socket.
3520032 //-----
3520033 // Flags to be set with:
3520034 // fcntl (fd, F_SETFD, ...);
3520035 //-----
3520036 #define FD_CLOEXEC 1 // Close the file
3520037 // descriptor upon
3520038 // execution of an
3520039 // exec() family
3520040 // function.
3520041 //-----

```

```

352042 // Values for type 'l_type', used for record locking
352043 // with 'fcntl()'.
352044 //-----
352045 #define F_RDLCK      0      // Read lock.
352046 #define F_WRLCK      1      // Write lock.
352047 #define F_UNLCK      2      // Remove lock.
352048 //-----
352049 // Flags for file creation, in place of 'oflag'
352050 // parameter for function 'open()'.
352051 //-----
352052 #define O_CREAT       000010 // Create file if it
352053 // does not exist.
352054 #define O_EXCL        000020 // Exclusive use flag.
352055 #define O_NOCTTY      000040 // Do not assign a
352056 // controlling
352057 // terminal.
352058 #define O_TRUNC       000100 // Truncation flag.
352059 //-----
352060 // Flags for the file status, used with 'open()' and
352061 // 'fcntl()'.
352062 //-----
352063 #define O_APPEND      000200 // Write append.
352064 #define O_DSYNC       000400 // Synchronized write
352065 // operations.
352066 #define O_NONBLOCK    001000 // Non-blocking mode.
352067 #define O_RSYNC       002000 // Synchronized read
352068 // operations.
352069 #define O_SYNC        004000 // Synchronized read
352070 // and write.
352071 //-----
352072 // File access mask selection.
352073 //-----
352074 #define O_ACCMODE     000003 // Mask to select the
352075 // last three bits,
352076 // used to specify the
352077 // main access
352078 // modes: read, write
352079 // and both.
352080 //-----
352081 // Main access modes.
352082 //-----
352083 #define O_RDONLY      000001 // Read.
352084 #define O_WRONLY      000002 // Write.
352085 #define O_RDWR       (O_RDONLY | O_WRONLY) // [1]
352086 //
352087 // [1] Both read and write.
352088 //
352089 //-----
352090 // Structure 'flock', used to file lock for POSIX
352091 // standard. It is not used inside os32.
352092 //-----
352093 struct flock
352094 {
352095     short int l_type; // Type of lock: F_RDLCK,
352096 // F_WRLCK, or F_UNLCK.
352097     short int l_whence; // Start reference point.
352098     off_t l_start; // Offset, from 'l_whence',
352099 // for the area start.
352100     off_t l_len; // Locked area size. Zero means up to
352101 // the end of the file.
352102     pid_t l_pid; // The process id blocking the area.
352103 };
352104 //-----
352105 // Function prototypes.
352106 //-----
352107 int creat (const char *path, mode_t mode);
352108 int fcntl (int fdn, int cmd, ...);
352109 int open (const char *path, int oflags, ...);
352110 //-----
352111
352112 #endif

```

95.6.1 lib/fcntl/creat.c ..... 774

95.6.2 lib/fcntl/fcntl.c ..... 775

95.6.3 lib/fcntl/open.c ..... 775

## 95.6.1 lib/fcntl/creat.c

«

Si veda la sezione 88.14.

```

3530001 #include <fcntl.h>
3530002 #include <sys/types.h>
3530003 //-----
3530004 int
3530005 creat (const char *path, mode_t mode)
3530006 {

```

```

3530007     return (open (path, O_WRONLY | O_CREAT | O_TRUNC, mode));
3530008 }

```

## 95.6.2 lib/fcntl/fcntl.c

Si veda la sezione 87.18.

«

```

3540001 #include <fcntl.h>
3540002 #include <stdarg.h>
3540003 #include <stddef.h>
3540004 #include <string.h>
3540005 #include <errno.h>
3540006 #include <sys/os32.h>
3540007 #include <limits.h>
3540008 //-----
3540009 int
3540010 fcntl (int fdn, int cmd, ...)
3540011 {
3540012     va_list ap;
3540013     sysmsg_fcntl_t msg;
3540014     va_start (ap, cmd);
3540015     //
3540016     // Well known arguments.
3540017     //
3540018     msg.fdn = fdn;
3540019     msg.cmd = cmd;
3540020     //
3540021     // Select other arguments.
3540022     //
3540023     switch (cmd)
3540024     {
3540025     case F_DUPFD:
3540026     case F_SETFD:
3540027     case F_SETFL:
3540028         msg.arg = va_arg (ap, int);
3540029         break;
3540030     case F_GETFD:
3540031     case F_GETFL:
3540032         break;
3540033     case F_GETOWN:
3540034     case F_SETOWN:
3540035     case F_GETLK:
3540036     case F_SETLK:
3540037     case F_SETLKW:
3540038         errset (E_NOT_IMPLEMENTED); // Not
3540039         // implemented.
3540040         return (-1);
3540041     default:
3540042         errset (EINVAL); // Not implemented.
3540043         return (-1);
3540044     }
3540045     //
3540046     // Do the system call.
3540047     //
3540048     sys (SYS_FCNTL, &msg, (sizeof msg));
3540049     errno = msg.errno;
3540050     errln = msg.errln;
3540051     strncpy (errfn, msg.errfn, PATH_MAX);
3540052     return (msg.ret);
3540053 }

```

## 95.6.3 lib/fcntl/open.c

Si veda la sezione 87.37.

«

```

3550001 #include <fcntl.h>
3550002 #include <stdarg.h>
3550003 #include <stddef.h>
3550004 #include <string.h>
3550005 #include <errno.h>
3550006 #include <sys/os32.h>
3550007 #include <limits.h>
3550008 //-----
3550009 int
3550010 open (const char *path, int oflags, ...)
3550011 {
3550012     va_list ap;
3550013     sysmsg_open_t msg;
3550014     va_start (ap, oflags);
3550015     msg.path = path;
3550016     msg.flags = oflags;
3550017     msg.mode = va_arg (ap, mode_t);
3550018     sys (SYS_OPEN, &msg, (sizeof msg));
3550019     errno = msg.errno;
3550020     errln = msg.errln;
3550021     strncpy (errfn, msg.errfn, PATH_MAX);
3550022     return (msg.ret);

```

```
350021 }
350022 }
```

## 95.7 os32: «lib/grp.h»

« Si veda la sezione 91.3.

```
350001 #ifndef _GRP_H
350002 #define _GRP_H 1
350003 //-----
350004 #include <restrict.h>
350005 #include <sys/types.h> // gid_t, uid_t
350006 //-----
350007 #define GR_MEM_MAX 32
350008 struct group
350009 {
350010     char *gr_name;
350011     char *gr_passwd;
350012     gid_t gr_gid;
350013     char *gr_mem[GR_MEM_MAX];
350014 };
350015 //-----
350016 struct group *getgrnt (void);
350017 void setgrnt (void);
350018 void endgrnt (void);
350019 struct group *getgrnam (const char *name);
350020 struct group *getgrgid (gid_t gid);
350021 //-----
350022 #endif
```

### 95.7.1 lib/grp/grent.c ..... 776

#### 95.7.1 lib/grp/grent.c

« Si veda la sezione 88.53.

```
357001 #include <grp.h>
357002 #include <stdio.h>
357003 #include <string.h>
357004 #include <stdlib.h>
357005 //-----
357006 static char buffer[BUFSIZ];
357007 static struct group gr;
357008 static FILE *fp = NULL;
357009 //-----
357010 struct group *
357011 getgrnt (void)
357012 {
357013     void *pstatus;
357014     char *char_gid;
357015     int i;
357016     //
357017     if (fp == NULL)
357018     {
357019         fp = fopen ("/etc/group", "r");
357020         if (fp == NULL)
357021         {
357022             return NULL;
357023         }
357024     }
357025     //
357026     pstatus = fgets (buffer, BUFSIZ, fp);
357027     if (pstatus == NULL)
357028     {
357029         return (NULL);
357030     }
357031     //
357032     // The parse is made with 'strtok()'. Please notice
357033     // that
357034     // 'strtok()' will not parse a line like the
357035     // following:
357036     // user:233:
357037     // The password field *must* have something,
357038     // otherwise the
357039     // GID will take the password place.
357040     // 'strtok()' will consider ':' the same as ':'!
357041     //
357042     gr.gr_name = strtok (buffer, ":");
357043     gr.gr_passwd = strtok (NULL, ":");
357044     char_gid = strtok (NULL, ":");
357045     for (i = 0; i < GR_MEM_MAX; i++)
357046     {
357047         gr.gr_mem[i] = strtok (NULL, "\n");
357048     }
357049     gr.gr_gid = (gid_t) atoi (char_gid);
357050     //
357051     return (&gr);
357052 }
```

```
357053 //-----
357054 void
357055 endgrnt (void)
357056 {
357057     int status;
357058     //
357059     if (fp != NULL)
357060     {
357061         status = fclose (fp);
357062         if (status != 0)
357063         {
357064             perror (NULL);
357065             fp = NULL;
357066         }
357067     }
357068 }
357069 //-----
357070 void
357071 setgrnt (void)
357072 {
357073     if (fp != NULL)
357074     {
357075         rewind (fp);
357076     }
357077 }
357078 //-----
357079 struct group *
357080 getgrnam (const char *name)
357081 {
357082     struct group *gr;
357083     //
357084     setgrnt ();
357085     //
357086     for (;;)
357087     {
357088         gr = getgrnt ();
357089         if (gr == NULL)
357090         {
357091             return (NULL);
357092         }
357093         if (strcmp (gr->gr_name, name) == 0)
357094         {
357095             return (gr);
357096         }
357097     }
357098 }
357099 //-----
357100 struct group *
357101 getgrgid (gid_t gid)
357102 {
357103     struct group *gr;
357104     //
357105     setgrnt ();
357106     //
357107     for (;;)
357108     {
357109         gr = getgrnt ();
357110         if (gr == NULL)
357111         {
357112             return (NULL);
357113         }
357114         if (gr->gr_gid == gid)
357115         {
357116             return (gr);
357117         }
357118     }
357119 }
357120 //-----
357121 }
357122 }
```

## 95.8 os32: «lib/inttypes.h»

« Si veda la sezione 91.3.

```
358001 #ifndef _INTTYPES_H
358002 #define _INTTYPES_H 1
358003 //-----
358004 #include <stdint.h>
358005 #include <wchar_t.h>
358006 #include <restrict.h>
358007 //-----
358008 typedef struct
358009 {
358010     intmax_t quot;
358011     intmax_t rem;
```

«

```

3580012 } imaxdiv_t;
3580013 //
3580014 imaxdiv_t imaxdiv (intmax_t numer, intmax_t denom);
3580015 //-----
3580016 // Output typesetting.
3580017 //-----
3580018 #define PRId8      "d"
3580019 #define PRId16     "d"
3580020 #define PRId32     "d"
3580021 #define PRId64     "lld"
3580022 //
3580023 #define PRIdLEAST8 "d"
3580024 #define PRIdLEAST16 "d"
3580025 #define PRIdLEAST32 "d"
3580026 #define PRIdLEAST64 "lld"
3580027 //
3580028 #define PRIdFAST8  "d"
3580029 #define PRIdFAST16 "d"
3580030 #define PRIdFAST32 "d"
3580031 #define PRIdFAST64 "lld"
3580032 //
3580033 #define PRIdMAX    "lld"
3580034 #define PRIdPTR    "d"
3580035 //
3580036 #define PRIi8      "i"
3580037 #define PRIi16     "i"
3580038 #define PRIi32     "i"
3580039 #define PRIi64     "lli"
3580040 //
3580041 #define PRIiLEAST8 "i"
3580042 #define PRIiLEAST16 "i"
3580043 #define PRIiLEAST32 "i"
3580044 #define PRIiLEAST64 "lli"
3580045 //
3580046 #define PRIiFAST8  "i"
3580047 #define PRIiFAST16 "i"
3580048 #define PRIiFAST32 "i"
3580049 #define PRIiFAST64 "lli"
3580050 //
3580051 #define PRIiMAX    "lli"
3580052 #define PRIiPTR    "i"
3580053 //
3580054 #define PRId8      "b" // PRId... is not
3580055 // standard!
3580056 #define PRId16     "b" //
3580057 #define PRId32     "b" //
3580058 #define PRId64     "llb" //
3580059 //
3580060 #define PRIdLEAST8 "b" //
3580061 #define PRIdLEAST16 "b" //
3580062 #define PRIdLEAST32 "b" //
3580063 #define PRIdLEAST64 "llb" //
3580064 //
3580065 #define PRIdFAST8  "b" //
3580066 #define PRIdFAST16 "b" //
3580067 #define PRIdFAST32 "b" //
3580068 #define PRIdFAST64 "llb" //
3580069 //
3580070 #define PRIdMAX    "llb" //
3580071 #define PRIdPTR    "b" //
3580072 //
3580073 #define PRIdO8     "o"
3580074 #define PRIdO16    "o"
3580075 #define PRIdO32    "o"
3580076 #define PRIdO64    "llo"
3580077 //
3580078 #define PRIdOLEAST8 "o"
3580079 #define PRIdOLEAST16 "o"
3580080 #define PRIdOLEAST32 "o"
3580081 #define PRIdOLEAST64 "llo"
3580082 //
3580083 #define PRIdOFAST8 "o"
3580084 #define PRIdOFAST16 "o"
3580085 #define PRIdOFAST32 "o"
3580086 #define PRIdOFAST64 "llo"
3580087 //
3580088 #define PRIdOMAX   "llo"
3580089 #define PRIdOPTR   "o"
3580090 //
3580091 #define PRIu8      "u"
3580092 #define PRIu16     "u"
3580093 #define PRIu32     "u"
3580094 #define PRIu64     "llu"
3580095 //
3580096 #define PRIuLEAST8 "u"
3580097 #define PRIuLEAST16 "u"
3580098 #define PRIuLEAST32 "u"

```

```

3580099 #define PRIuLEAST64 "llu"
3580100 //
3580101 #define PRIuFAST8  "u"
3580102 #define PRIuFAST16 "u"
3580103 #define PRIuFAST32 "u"
3580104 #define PRIuFAST64 "llu"
3580105 //
3580106 #define PRIuMAX    "llu"
3580107 #define PRIuPTR    "u"
3580108 //
3580109 #define PRIx8      "x"
3580110 #define PRIx16     "x"
3580111 #define PRIx32     "x"
3580112 #define PRIx64     "llx"
3580113 //
3580114 #define PRIxLEAST8 "x"
3580115 #define PRIxLEAST16 "x"
3580116 #define PRIxLEAST32 "x"
3580117 #define PRIxLEAST64 "llx"
3580118 //
3580119 #define PRIxFAST8  "x"
3580120 #define PRIxFAST16 "x"
3580121 #define PRIxFAST32 "x"
3580122 #define PRIxFAST64 "llx"
3580123 //
3580124 #define PRIxMAX    "llx"
3580125 #define PRIxPTR    "x"
3580126 //
3580127 #define PRIX8      "X"
3580128 #define PRIX16     "X"
3580129 #define PRIX32     "X"
3580130 #define PRIX64     "llX"
3580131 //
3580132 #define PRIXLEAST8 "X"
3580133 #define PRIXLEAST16 "X"
3580134 #define PRIXLEAST32 "X"
3580135 #define PRIXLEAST64 "llX"
3580136 //
3580137 #define PRIXFAST8  "X"
3580138 #define PRIXFAST16 "X"
3580139 #define PRIXFAST32 "X"
3580140 #define PRIXFAST64 "llX"
3580141 //
3580142 #define PRIXMAX    "llX"
3580143 #define PRIXPTR    "X"
3580144 //-----
3580145 // Input scan and evaluation.
3580146 //-----
3580147 #define SCNd8      "hhd"
3580148 #define SCNd16     "hd"
3580149 #define SCNd32     "d"
3580150 #define SCNd64     "lld"
3580151 //
3580152 #define SCNdLEAST8 "hhd"
3580153 #define SCNdLEAST16 "hd"
3580154 #define SCNdLEAST32 "d"
3580155 #define SCNdLEAST64 "lld"
3580156 //
3580157 #define SCNdFAST8  "hhd"
3580158 #define SCNdFAST16 "d"
3580159 #define SCNdFAST32 "d"
3580160 #define SCNdFAST64 "lld"
3580161 //
3580162 #define SCNdMAX    "lld"
3580163 #define SCNdPTR    "d"
3580164 //
3580165 #define SCNi8      "hhi"
3580166 #define SCNi16     "hi"
3580167 #define SCNi32     "i"
3580168 #define SCNi64     "lli"
3580169 //
3580170 #define SCNiLEAST8 "hhi"
3580171 #define SCNiLEAST16 "hi"
3580172 #define SCNiLEAST32 "i"
3580173 #define SCNiLEAST64 "lli"
3580174 //
3580175 #define SCNiFAST8  "hhi"
3580176 #define SCNiFAST16 "i"
3580177 #define SCNiFAST32 "i"
3580178 #define SCNiFAST64 "lli"
3580179 //
3580180 #define SCNiMAX    "lli"
3580181 #define SCNiPTR    "i"
3580182 //
3580183 #define SCNb8      "hhb" // SCNb... is not
3580184 // standard!
3580185 #define SCNb16     "hb" //

```



```

3580186 #define SCNb32      "b" //
3580187 #define SCNb64      "llb" //
3580188 //
3580189 #define SCNbLEAST8   "hbb" //
3580190 #define SCNbLEAST16  "hb" //
3580191 #define SCNbLEAST32  "b" //
3580192 #define SCNbLEAST64  "llb" //
3580193 //
3580194 #define SCNbFAST8    "hbb" //
3580195 #define SCNbFAST16   "b" //
3580196 #define SCNbFAST32   "b" //
3580197 #define SCNbFAST64   "llb" //
3580198 //
3580199 #define SCNbMAX       "llb" //
3580200 #define SCNbPTR       "b" //
3580201 //
3580202 #define SCNo8         "hho"
3580203 #define SCNo16        "ho"
3580204 #define SCNo32        "o"
3580205 #define SCNo64        "llo"
3580206 //
3580207 #define SCNoLEAST8    "hho"
3580208 #define SCNoLEAST16   "ho"
3580209 #define SCNoLEAST32   "o"
3580210 #define SCNoLEAST64   "llo"
3580211 //
3580212 #define SCNoFAST8     "hho"
3580213 #define SCNoFAST16    "o"
3580214 #define SCNoFAST32    "o"
3580215 #define SCNoFAST64    "llo"
3580216 //
3580217 #define SCNoMAX       "llo"
3580218 #define SCNoPTR       "o"
3580219 //
3580220 #define SCNu8         "hhu"
3580221 #define SCNu16        "hu"
3580222 #define SCNu32        "u"
3580223 #define SCNu64        "llu"
3580224 //
3580225 #define SCNuLEAST8    "hhu"
3580226 #define SCNuLEAST16   "hu"
3580227 #define SCNuLEAST32   "u"
3580228 #define SCNuLEAST64   "llu"
3580229 //
3580230 #define SCNuFAST8     "hhu"
3580231 #define SCNuFAST16    "u"
3580232 #define SCNuFAST32    "u"
3580233 #define SCNuFAST64    "llu"
3580234 //
3580235 #define SCNuMAX       "llu"
3580236 #define SCNuPTR       "u"
3580237 //
3580238 #define SCNx8         "hhx"
3580239 #define SCNx16        "hx"
3580240 #define SCNx32        "x"
3580241 #define SCNx64        "llx"
3580242 //
3580243 #define SCNxLEAST8    "hhx"
3580244 #define SCNxLEAST16   "hx"
3580245 #define SCNxLEAST32   "x"
3580246 #define SCNxLEAST64   "llx"
3580247 //
3580248 #define SCNxFAST8     "hhx"
3580249 #define SCNxFAST16    "x"
3580250 #define SCNxFAST32    "x"
3580251 #define SCNxFAST64    "llx"
3580252 //
3580253 #define SCNxMAX       "llx"
3580254 #define SCNxPTR       "x"
3580255 //-----
3580256 intmax_t imaxabs (intmax_t j);
3580257 intmax_t strtouimax (const char *restrict nptr,
3580258                     char **restrict endptr, int base);
3580259 uintmax_t strtouimax (const char *restrict nptr,
3580260                      char **restrict endptr, int base);
3580261 intmax_t wcstouimax (const wchar_t * restrict nptr,
3580262                     wchar_t ** restrict endptr, int base);
3580263 uintmax_t wcstouimax (const wchar_t * restrict nptr,
3580264                      wchar_t ** restrict endptr, int base);
3580265 //-----
3580266 #endif

```

95.8.1 lib/inttypes/imaxabs.c ..... 781

95.8.2 lib/inttypes/imaxdiv.c ..... 781

## 95.8.1 lib/inttypes/imaxabs.c

Si veda la sezione 88.3.

```

3590001 #include <inttypes.h>
3590002 //-----
3590003 intmax_t
3590004 imaxabs (intmax_t j)
3590005 {
3590006     if (j < 0)
3590007     {
3590008         return -j;
3590009     }
3590010     else
3590011     {
3590012         return j;
3590013     }
3590014 }

```

## 95.8.2 lib/inttypes/imaxdiv.c

Si veda la sezione 88.17.

```

3600001 #include <inttypes.h>
3600002 //-----
3600003 imaxdiv_t
3600004 imaxdiv (intmax_t numer, intmax_t denom)
3600005 {
3600006     imaxdiv_t d;
3600007     d.quot = numer / denom;
3600008     d.rem = numer % denom;
3600009     return d;
3600010 }

```

## 95.9 os32: «lib/libgen.h»

Si veda la sezione 91.3.

```

3610001 #ifndef _LIBGEN_H
3610002 #define _LIBGEN_H      1
3610003
3610004 //-----
3610005 char *basename (char *path);
3610006 char *dirname (char *path);
3610007 //-----
3610008
3610009 #endif

```

95.9.1 lib/libgen/basename.c ..... 781

95.9.2 lib/libgen/dirname.c ..... 782

## 95.9.1 lib/libgen/basename.c

Si veda la sezione 88.10.

```

3620001 #include <libgen.h>
3620002 #include <limits.h>
3620003 #include <stddef.h>
3620004 #include <string.h>
3620005 //-----
3620006 char *
3620007 basename (char *path)
3620008 {
3620009     static char *point = "."; // When 'path' is
3620010 // NULL.
3620011     char *p; // Pointer inside 'path'.
3620012     int i; // Scan index inside 'path'.
3620013 //
3620014 // Empty path.
3620015 //
3620016     if (path == NULL || strlen (path) == 0)
3620017     {
3620018         return (point);
3620019     }
3620020 //
3620021 // Remove all final '/' if it exists, excluded the
3620022 // first character:
3620023 // 'i' is kept greater than zero.
3620024 //
3620025     for (i = (strlen (path) - 1);
3620026          i > 0 && path[i] == '/'; i--)
3620027     {
3620028         path[i] = 0;
3620029     }
3620030 //
3620031 // After removal of extra final '/', if there is
3620032 // only one '/', this

```

```

362033 // is to be returned.
362034 //
362035 if (strncmp (path, "/", PATH_MAX) == 0)
362036 {
362037     return (path);
362038 }
362039 //
362040 // If there are no '/'.
362041 //
362042 if (strchr (path, '/') == NULL)
362043 {
362044     return (path);
362045 }
362046 //
362047 // Find the last '/' and calculate a pointer to the
362048 // base name.
362049 //
362050 p = strrchr (path, (unsigned int) '/');
362051 p++;
362052 //
362053 // Return the pointer to the base name.
362054 //
362055 return (p);
362056 }

```

## 95.9.2 lib/libgen/dirname.c

« Si veda la sezione 88.10.

```

363001 #include <libgen.h>
363002 #include <limits.h>
363003 #include <stddef.h>
363004 #include <string.h>
363005 //-----
363006 char *
363007 dirname (char *path)
363008 {
363009     static char *point = "."; // When 'path' is
363010 // NULL.
363011     char *p; // Pointer inside 'path'.
363012     int i; // Scan index inside 'path'.
363013 //
363014 // Empty path.
363015 //
363016 if (path == NULL || strlen (path) == 0)
363017 {
363018     return (point);
363019 }
363020 //
363021 // Simple cases.
363022 //
363023 if (strncmp (path, "/", PATH_MAX) == 0 ||
363024     strncmp (path, ".", PATH_MAX) == 0 ||
363025     strncmp (path, "..", PATH_MAX) == 0)
363026 {
363027     return (path);
363028 }
363029 //
363030 // Remove all final '/' if it exists, excluded the
363031 // first character:
363032 // 'i' is kept greater than zero.
363033 //
363034 for (i = (strlen (path) - 1);
363035      i > 0 && path[i] == '/'; i--)
363036 {
363037     path[i] = 0;
363038 }
363039 //
363040 // After removal of extra final '/', if there is
363041 // only one '/', this
363042 // is to be returned.
363043 //
363044 if (strncmp (path, "/", PATH_MAX) == 0)
363045 {
363046     return (path);
363047 }
363048 //
363049 // If there are no '/'
363050 //
363051 if (strchr (path, '/') == NULL)
363052 {
363053     return (point);
363054 }
363055 //
363056 // If there is only a '/' a the beginning.
363057 //
363058 if (path[0] == '/' &&

```

```

363059     strchr (&path[1], (unsigned int) '/') == NULL)
363060 {
363061     path[1] = 0;
363062     return (path);
363063 }
363064 //
363065 // Replace the last '/' with zero.
363066 //
363067 p = strrchr (path, (unsigned int) '/');
363068 *p = 0;
363069 //
363070 // Now remove extra duplicated final '/', except the
363071 // very first
363072 // character: 'i' is kept greater than zero.
363073 //
363074 for (i = (strlen (path) - 1);
363075      i > 0 && path[i] == '/'; i--)
363076 {
363077     path[i] = 0;
363078 }
363079 //
363080 // Now 'path' appears as a reduced string: the
363081 // original path string
363082 // is modified.
363083 //
363084 return (path);
363085 }

```

## 95.10 os32: «lib/netinet/icmp.h»

Si veda la sezione 91.3. »

```

364001 #ifndef __NETINET_ICMP_H
364002 #define __NETINET_ICMP_H 1
364003 //-----
364004 // GNU C compatible ICMPv4 header and definitions
364005 //-----
364006 #include <sys/types.h>
364007 #include <netinet/in.h>
364008 #include <netinet/ip.h>
364009 //-----
364010 struct icmp_hdr
364011 {
364012     uint8_t type; // message type [1]
364013     uint8_t code; // type sub-code [2]
364014     uint16_t checksum;
364015     union
364016     {
364017         struct
364018         {
364019             uint16_t id;
364020             uint16_t sequence;
364021         } __attribute__ ((packed)) echo; // echo
364022         // datagram
364023         uint32_t gateway; // gateway address
364024         struct
364025         {
364026             uint16_t unused;
364027             uint16_t mtu;
364028         } __attribute__ ((packed)) frag; // path mtu
364029         // discovery
364030     } un;
364031 } __attribute__ ((packed));
364032 //
364033 // [1] message type:
364034 //
364035 #define ICMP_ECHOREPLY 0 // echo reply
364036 #define ICMP_DEST_UNREACH 3 // destination
364037 // unreachable
364038 #define ICMP_SOURCE_QUENCH 4 // source
364039 // quench
364040 #define ICMP_REDIRECT 5 // redirect
364041 // (change
364042 // route)
364043 #define ICMP_ECHO 8 // echo
364044 // request
364045 #define ICMP_TIME_EXCEEDED 11 // time
364046 // exceeded
364047 #define ICMP_PARAMETERPROB 12 // parameter
364048 // problem
364049 #define ICMP_TIMESTAMP 13 // timestamp
364050 // request
364051 #define ICMP_TIMESTAMPREPLY 14 // timestamp
364052 // reply
364053 #define ICMP_INFO_REQUEST 15 // information
364054 // request
364055 #define ICMP_INFO_REPLY 16 // information

```

```

3640056 // reply
3640057 #define ICMP_ADDRESS 17 // address
3640058 // mask
3640059 // request
3640060 #define ICMP_ADDRESSREPLY 18 // address
3640061 // mask reply
3640062 #define NR_ICMP_TYPES 18
3640063 //
3640064 // [2] type ICMP_DEST_UNREACH, code:
3640065 //
3640066 #define ICMP_NET_UNREACH 0 // network
3640067 // unreachable
3640068 #define ICMP_HOST_UNREACH 1 // host
3640069 // unreachable
3640070 #define ICMP_PROT_UNREACH 2 // protocol
3640071 // unreachable
3640072 #define ICMP_PORT_UNREACH 3 // port
3640073 // unreachable
3640074 #define ICMP_FRAG_NEEDED 4 // fragmentation
3640075 // needed/DF
3640076 // set
3640077 #define ICMP_SR_FAILED 5 // source
3640078 // route
3640079 // failed
3640080 #define ICMP_NET_UNKOWN 6 // destination
3640081 // network
3640082 // unknown
3640083 #define ICMP_HOST_UNKOWN 7 // destination
3640084 // host
3640085 // unknown
3640086 #define ICMP_HOST_ISOLATED 8 // source host
3640087 // isolated
3640088 #define ICMP_NET_ANO 9 // destination
3640089 // network
3640090 // administratively
3640091 // prohibited
3640092 #define ICMP_HOST_ANO 10 // destination
3640093 // host
3640094 // administratively
3640095 // prohibited
3640096 #define ICMP_NET_UNR_TOS 11 // network
3640097 // unreachable
3640098 // for this
3640099 // type of
3640100 // service
3640101 #define ICMP_HOST_UNR_TOS 12 // host
3640102 // unreachable
3640103 // for this
3640104 // type of
3640105 // service
3640106 #define ICMP_PKT_FILTERED 13 // packet
3640107 // filtered
3640108 #define ICMP_PREC_VIOLATION 14 // precedence
3640109 // violation
3640110 #define ICMP_PREC_CUTOFF 15 // precedence
3640111 // cut off
3640112 #define NR_ICMP_UNREACH 15 // instead of
3640113 // hardcoding
3640114 // immediate
3640115 // value
3640116 //
3640117 // [2] type ICMP_REDIRECT, code:
3640118 //
3640119 #define ICMP_REDIRECT_NET 0 // redirect
3640120 // net
3640121 #define ICMP_REDIRECT_HOST 1 // redirect
3640122 // host
3640123 #define ICMP_REDIRECT_NETTOS 2 // redirect
3640124 // net for TOS
3640125 #define ICMP_REDIRECT_HOSTTOS 3 // redirect
3640126 // host for
3640127 // TOS
3640128 //
3640129 // [2] type ICMP_TIME_EXCEEDED, code:
3640130 //
3640131 #define ICMP_EXC_TTL 0 // TTL count
3640132 // exceeded
3640133 #define ICMP_EXC_FRAGTIME 1 // fragment
3640134 // reasm time
3640135 // exceeded
3640136 //-----
3640137 #endif

```

## 95.11 os32: «lib/netinet/in.h»

Si veda la sezione 91.3.

```

3600001 #ifndef _NETINET_IN_H
3600002 #define _NETINET_IN_H 1
3600003 //-----
3600004 #include <stdint.h>
3600005 #include <sys/sa_family_t.h>
3600006 //-----
3600007 typedef uint16_t in_port_t; // Port number. [1]
3600008 typedef uint32_t in_addr_t; // IPv4 address.
3600009 //
3600010 // [1] Types 'in_port_t' and 'in_addr_t' are to be
3600011 // intended for network byte order IPv4 integer
3600012 // address, at least because this type is
3600013 // used inside the type 'struct in_addr', that is
3600014 // surely in network byte order. But attention must
3600015 // be made to mistakes: for example,
3600016 // inside the file <netinet/in.h> from GNU sources,
3600017 // there are some macro defining default netmask
3600018 // like this:
3600019 //
3600020 // #define IN_CLASSA(a)
3600021 // (((in_addr_t)(a) & 0x80000000) == 0)
3600022 // #define IN_CLASSB(a)
3600023 // (((in_addr_t)(a) & 0xc0000000) == 0x80000000)
3600024 // #define IN_CLASSC(a)
3600025 // (((in_addr_t)(a) & 0xe0000000) == 0xc0000000)
3600026 //
3600027 // Such macro can work only if the architecture is
3600028 // big-endian.
3600029 //
3600030 //-----
3600031 //
3600032 // IPv4 address.
3600033 //
3600034 struct in_addr
3600035 {
3600036     in_addr_t s_addr;
3600037 };
3600038 //
3600039 // struct sockaddr_in, members in *network*byte*order*.
3600040 //
3600041 struct sockaddr_in
3600042 {
3600043     sa_family_t sin_family; // AF_INET.
3600044     in_port_t sin_port; // Port number.
3600045     struct in_addr sin_addr; // IP address.
3600046     uint8_t sin_zero[8]; // [2]
3600047 };
3600048 //
3600049 // [2] The type 'struct sockaddr_in' must be
3600050 // replaceable with the type 'struct sockaddr',
3600051 // with a cast. So it is necessary to fill the
3600052 // unused space with a filler.
3600053 //
3600054 //-----
3600055 //
3600056 // IPv6 address, network byte order.
3600057 //
3600058 struct in6_addr
3600059 {
3600060     uint8_t s6_addr[16];
3600061 };
3600062 //
3600063 // struct sockaddr_in6, members in network byte order.
3600064 //
3600065 struct sockaddr_in6
3600066 {
3600067     sa_family_t sin6_family; // AF_INET6.
3600068     in_port_t sin6_port; // Port number.
3600069     uint32_t sin6_flowinfo; // IPv6 traffic class
3600070 // and flow info.
3600071     struct in6_addr sin6_addr; // IPv6 address.
3600072     uint32_t sin6_scope_id; // Set of interfaces
3600073 // for a scope.
3600074 };
3600075 //-----
3600076 //external in6_addr in6addr_any;
3600077 // #define IN6ADDR_ANY_INIT ...
3600078 //external struct in6_addr in6addr_loopback;
3600079 // #define IN6ADDR_LOOPBACK_INIT ...
3600080 //-----
3600081 //
3600082 //
3600083 //
3600084 struct ipv6_mreq

```

```

3650085 {
3650086     struct in6_addr ipv6mr_multiaddr;    // IPv6
3650087     // multicast
3650088     // address.
3650089     unsigned int ipv6mr_interface;      // Interface
3650090     // index.
3650091 };
3650092 //-----
3650093 #define IPPROTO_IP      0      // Internet protocol.
3650094 #define IPPROTO_ICMP    1      // Contro message
3650095     // protocol.
3650096 #define IPPROTO_TCP     6      // Transmission
3650097     // control protocol.
3650098 #define IPPROTO_UDP     17     // User datagram
3650099     // protocol.
3650100 #define IPPROTO_IPV6    41     // Internet protocol
3650101     // version 6.
3650102 #define IPPROTO_RAW     255    // Raw IP packets
3650103     // protocol
3650104 //-----
3650105 //
3650106 // 0.0.0.0
3650107 //
3650108 #define INADDR_ANY      ((in_addr_t) 0x00000000)
3650109 //
3650110 // 255.255.255.255
3650111 //
3650112 #define INADDR_BROADCAST ((in_addr_t) 0xffffffff)
3650113 //
3650114 // 127.0.0.1
3650115 //
3650116 #define INADDR_LOOPBACK ((in_addr_t) 0x7f000001)
3650117 //
3650118 //
3650119 //
3650120 #define INET_ADDRSTRLEN 16     // IPv4 address string
3650121     // size.
3650122 #define INET6_ADDRSTRLEN 46    // IPv6 address string
3650123     // size.
3650124 //-----
3650125 #endif

```

## 95.12 os32: «lib/netinet/ip.h»

« Si veda la sezione 91.3.

```

3660001 #ifndef _NETINET_IP_H
3660002 #define _NETINET_IP_H      1
3660003 //-----
3660004 // GNU C compatible IPv4 header.
3660005 //-----
3660006 #include <netinet/in.h>
3660007 //-----
3660008 struct iphdr
3660009 {
3660010     uint16_t ihl:4,          // header length / 4
3660011     version:4;             // IP version
3660012     uint8_t  tos;           // type of service
3660013     uint16_t tot_len;       // total packet length
3660014     uint16_t id;            // identification
3660015     uint16_t frag_off;      // fragment offset field
3660016     uint8_t  ttl;          // time to live
3660017     uint8_t  protocol;     // contained protocol
3660018     uint16_t check;        // header checksum
3660019     in_addr_t saddr;       // source IP address
3660020     in_addr_t daddr;       // destination IP address
3660021     //
3660022     // Options after this point.
3660023     //
3660024 };
3660025 //-----
3660026 #define IPVERSION      4      // IP version number
3660027 #define IP_MAXPACKET   65535  // maximum packet size
3660028 //
3660029 #define MAXTTL         255    // maximum time to
3660030     // live (seconds)
3660031 #define IPDEFTTL      64     // default ttl, from
3660032     // RFC 1340
3660033 #define IPFRAGTTL     60     // time to live for
3660034     // fragments
3660035 #define IPTTLDEC       1     // subtracted when
3660036     // forwarding
3660037 //
3660038 #define IP_MSS         576    // default maximum
3660039     // segment size

```

```

3660040 //-----
3660041 #endif

```

## 95.13 os32: «lib/netinet/tcp.h»

« Si veda la sezione 91.3.

```

3670001 #ifndef _NETINET_TCP_H
3670002 #define _NETINET_TCP_H    1
3670003 //-----
3670004 // GNU C compatible UDP header.
3670005 //-----
3670006 #include <sys/types.h>
3670007 //-----
3670008 struct tcphdr
3670009 {
3670010     uint16_t source;
3670011     uint16_t dest;
3670012     uint32_t seq;
3670013     uint32_t ack_seq;
3670014     uint16_t resl:4,
3670015     doff:4,
3670016     fin:1, syn:1, rst:1, psh:1, ack:1, urg:1, res2:2;
3670017     uint16_t window;
3670018     uint16_t check;
3670019     uint16_t urg_ptr;
3670020 };
3670021 //-----
3670022 // ATTENZIONE: per dare un significato allo stato di
3670023 // una connessione, occorre distinguere in che modo si
3670024 // trova inizialmente il socket:
3670025 // attivo o passivo (passivo quando rimane in ascolto
3670026 // per una connessione).
3670027 //
3670028 enum
3670029 {
3670030     TCP_LISTEN = 1,         // waiting a connection
3670031     // request
3670032     TCP_SYN_SENT,         // SYN was sent, waiting from the
3670033     // response SYN
3670034     TCP_SYN_RECV,        // SYN received, waiting for ACK
3670035     TCP_ESTABLISHED,     // SYN sent, SYN received and
3670036     // ACK sent
3670037     TCP_FIN_WAIT1,       // local close, FIN sent,
3670038     // waiting ACK or FIN
3670039     TCP_FIN_WAIT2,       // FIN sent, ACK received,
3670040     // waiting FIN
3670041     TCP_CLOSE_WAIT,      // FIN received, ACK sent,
3670042     // waiting local close
3670043     TCP_CLOSING,         // FIN sent, FIN received, ACK sent,
3670044     // waiting ACK
3670045     TCP_LAST_ACK,        // FIN received, ACK and FIN sent,
3670046     // waiting ACK
3670047     TCP_TIME_WAIT,       // after TCP_LAST_ACK, wait a
3670048     // little and remove
3670049     TCP_CLOSE,           // connection removed
3670050     TCP_RESET            // connection reset (not standard)
3670051 };
3670052 //-----
3670053 #define TCPOPT_EOL      0
3670054 #define TCPOPT_NOP      1
3670055 #define TCPOPT_MAXSEG   2
3670056 #define TCPOLEN_MAXSEG  4
3670057 #define TCPOPT_WINDOW   3
3670058 #define TCPOLEN_WINDOW  3
3670059 #define TCPOPT_SACK_PERMITTED 4
3670060 #define TCPOLEN_SACK_PERMITTED 2
3670061 #define TCPOPT_SACK     5
3670062 #define TCPOPT_TIMESTAMP 8
3670063 #define TCPOLEN_TIMESTAMP 10
3670064 //-----
3670065 //
3670066 // TCP max segment size: IP_MSS - IP header size.
3670067 // Suppose to have a max IP header of 56 bytes,
3670068 // TCP_MSS == 520.
3670069 //
3670070 #define TCP_MSS         520
3670071 //-----
3670072 // LA STRUTTURA SEGUENTE È DA VALUTARE, forse conviene
3670073 // fare una tabella a parte per le connessioni TCP.
3670074 //
3670075 struct tcp_info
3670076 {
3670077     uint8_t tcpi_state;
3670078     uint8_t tcpi_ca_state;

```

```

3670079 uint8_t tcpi_retransmits;
3670080 uint8_t tcpi_probes;
3670081 uint8_t tcpi_backoff;
3670082 uint8_t tcpi_options;
3670083 uint8_t tcpi_snd_wscale:4, tcpi_rcv_wscale:4;
3670084
3670085 uint32_t tcpi_rto;
3670086 uint32_t tcpi_ato;
3670087 uint32_t tcpi_snd_mss;
3670088 uint32_t tcpi_rcv_mss;
3670089
3670090 uint32_t tcpi_unacked;
3670091 uint32_t tcpi_sacked;
3670092 uint32_t tcpi_lost;
3670093 uint32_t tcpi_retrans;
3670094 uint32_t tcpi_fackets;
3670095
3670096 /* Times. */
3670097 uint32_t tcpi_last_data_sent;
3670098
3670099 /* Not remembered, sorry. */
3670100 uint32_t tcpi_last_ack_sent;
3670101
3670102 uint32_t tcpi_last_data_rcv;
3670103 uint32_t tcpi_last_ack_rcv;
3670104
3670105 /* Metrics. */
3670106 uint32_t tcpi_pmtu;
3670107 uint32_t tcpi_rcv_ssthresh;
3670108 uint32_t tcpi_rtt;
3670109 uint32_t tcpi_rttvar;
3670110 uint32_t tcpi_snd_ssthresh;
3670111 uint32_t tcpi_snd_cwnd;
3670112 uint32_t tcpi_advms;
3670113 uint32_t tcpi_reordering;
3670114
3670115 uint32_t tcpi_rcv_rtt;
3670116 uint32_t tcpi_rcv_space;
3670117
3670118 uint32_t tcpi_total_retrans;
3670119 };
3670120
3670121
3670122 //-----
3670123 #endif

```

## 95.14 os32: «lib/netinet/udp.h»

« Si veda la sezione 91.3.

```

3680001 #ifndef __NETINET_UDP_H
3680002 #define __NETINET_UDP_H 1
3680003 //-----
3680004 // GNU C compatible UDP header.
3680005 //-----
3680006 #include <sys/types.h>
3680007 //-----
3680008 struct udphdr
3680009 {
3680010     uint16_t source; // source port
3680011     uint16_t dest; // destination port
3680012     uint16_t len; // length
3680013     uint16_t check; // checksum
3680014 } __attribute__((packed));
3680015 //-----
3680016 #endif

```

## 95.15 os32: «lib/pwd.h»

« Si veda la sezione 91.3.

```

3690001 #ifndef _PWD_H
3690002 #define _PWD_H 1
3690003 //-----
3690004 #include <restrict.h>
3690005 #include <sys/types.h> // gid_t, uid_t
3690006 //-----
3690007 struct passwd
3690008 {
3690009     char *pw_name;
3690010     char *pw_passwd;
3690011     uid_t pw_uid;

```

```

3690012     gid_t pw_gid;
3690013     char *pw_gecos;
3690014     char *pw_dir;
3690015     char *pw_shell;
3690016 };
3690017 //-----
3690018 struct passwd *getpwent (void);
3690019 void setpwent (void);
3690020 void endpwent (void);
3690021 struct passwd *getpwnam (const char *name);
3690022 struct passwd *getpwuid (uid_t uid);
3690023 //-----
3690024
3690025 #endif

```

## 95.15.1 lib/pwd/pwent.c ..... 789

### 95.15.1 lib/pwd/pwent.c

« Si veda la sezione 88.57.

```

3700001 #include <pwd.h>
3700002 #include <stdio.h>
3700003 #include <string.h>
3700004 #include <stdlib.h>
3700005 //-----
3700006 static char buffer[BUFSIZ];
3700007 static struct passwd pw;
3700008 static FILE *fp = NULL;
3700009 //-----
3700010 struct passwd *
3700011 getpwent (void)
3700012 {
3700013     void *pstatus;
3700014     char *char_uid;
3700015     char *char_gid;
3700016     //
3700017     if (fp == NULL)
3700018     {
3700019         fp = fopen ("/etc/passwd", "r");
3700020         if (fp == NULL)
3700021         {
3700022             return NULL;
3700023         }
3700024     }
3700025     //
3700026     pstatus = fgets (buffer, BUFSIZ, fp);
3700027     if (pstatus == NULL)
3700028     {
3700029         return (NULL);
3700030     }
3700031     //
3700032     // The parse is made with 'strtok()'. Please notice
3700033     // that
3700034     // 'strtok()' will not parse a line like the
3700035     // following:
3700036     // user::1001:233:...
3700037     // The password field *must* have something,
3700038     // otherwise the
3700039     // UID will take the password place.
3700040     // 'strtok()' will consider ':' the same as ':'!
3700041     //
3700042     pw.pw_name = strtok (buffer, ":");
3700043     pw.pw_passwd = strtok (NULL, ":");
3700044     char_uid = strtok (NULL, ":");
3700045     char_gid = strtok (NULL, ":");
3700046     pw.pw_gecos = strtok (NULL, ":");
3700047     pw.pw_dir = strtok (NULL, ":");
3700048     pw.pw_shell = strtok (NULL, "\n");
3700049     pw.pw_uid = (uid_t) atoi (char_uid);
3700050     pw.pw_gid = (gid_t) atoi (char_gid);
3700051     //
3700052     return (&pw);
3700053 }
3700054 //-----
3700055 void
3700056 endpwent (void)
3700057 {
3700058     int status;
3700059     //
3700060     if (fp != NULL)
3700061     {
3700062         status = fclose (fp);
3700063         if (status != 0)
3700064         {
3700065             perror (NULL);

```

```

370067     fp = NULL;
370068     }
370069     else
370070     {
370071         ; // printf ("%s] fclose (fp)\n",
370072         // __func__);
370073     }
370074     }
370075 }
370076
370077 //-----
370078 void
370079 setpwent (void)
370080 {
370081     if (fp != NULL)
370082     {
370083         rewind (fp);
370084     }
370085 }
370086
370087 //-----
370088 struct passwd *
370089 getpwnam (const char *name)
370090 {
370091     struct passwd *pw;
370092     //
370093     setpwent ();
370094     //
370095     for (;;)
370096     {
370097         pw = getpwent ();
370098         if (pw == NULL)
370099         {
370100             return (NULL);
370101         }
370102         if (strcmp (pw->pw_name, name) == 0)
370103         {
370104             return (pw);
370105         }
370106     }
370107 }
370108
370109 //-----
370110 struct passwd *
370111 getpwuid (uid_t uid)
370112 {
370113     struct passwd *pw;
370114     //
370115     setpwent ();
370116     //
370117     for (;;)
370118     {
370119         pw = getpwent ();
370120         if (pw == NULL)
370121         {
370122             return (NULL);
370123         }
370124         if (pw->pw_uid == uid)
370125         {
370126             return (pw);
370127         }
370128     }
370129 }

```

## 95.16 os32: «lib/setjmp.h»

« Si veda la sezione 87.49.

```

3710001 #ifndef _SETJMP_H
3710002 #define _SETJMP_H 1
3710003 //-----
3710004 #include <sys/os32.h>
3710005 #include <NULL.h>
3710006 //-----
3710007 typedef struct
3710008 {
3710009     uint32_t eax0;
3710010     uint32_t ecx0;
3710011     uint32_t edx0;
3710012     uint32_t ebx0;
3710013     uint32_t ebp0;
3710014     uint32_t esi0;
3710015     uint32_t edi0;
3710016     uint32_t ds0;
3710017     uint32_t es0;
3710018     uint32_t fs0;
3710019     uint32_t gs0;

```

```

3710020     uint32_t eip0;
3710021     uint32_t cs0;
3710022     uint32_t eflags0;
3710023     //
3710024     uint32_t eip1;
3710025     uint32_t syscallnr;
3710026     uint32_t msg_pointer;
3710027     uint32_t msg_size;
3710028     //
3710029     uint32_t env;
3710030     uint32_t ret;
3710031     uint32_t ebp1;
3710032     uint32_t eip2;
3710033     //
3710034 } jmp_stack_t;
3710035
3710036 typedef struct
3710037 {
3710038     uint32_t esp0;
3710039     uint32_t eax0;
3710040     uint32_t ecx0;
3710041     uint32_t edx0;
3710042     uint32_t ebx0;
3710043     uint32_t ebp0;
3710044     uint32_t esi0;
3710045     uint32_t edi0;
3710046     uint32_t ds0;
3710047     uint32_t es0;
3710048     uint32_t fs0;
3710049     uint32_t gs0;
3710050     uint32_t eip0;
3710051     uint32_t cs0;
3710052     uint32_t eflags0;
3710053     //
3710054     uint32_t eip1;
3710055     uint32_t syscallnr;
3710056     uint32_t msg_pointer;
3710057     uint32_t msg_size;
3710058     //
3710059     uint32_t env;
3710060     uint32_t ret;
3710061     uint32_t ebp1;
3710062     uint32_t eip2;
3710063     //
3710064 } jmp_env_t;
3710065 //
3710066 typedef char jmp_buf[sizeof (jmp_env_t)];
3710067 //-----
3710068 int setjmp (jmp_buf env);
3710069 void longjmp (jmp_buf env, int val);
3710070 //-----
3710071 #endif

```

95.16.1 lib/setjmp/longjmp.c ..... 791

95.16.2 lib/setjmp/setjmp.s ..... 791

95.16.1 lib/setjmp/longjmp.c

« Si veda la sezione 87.49.

```

3720001 #include <sys/os32.h>
3720002 #include <setjmp.h>
3720003 //-----
3720004 void
3720005 longjmp (jmp_buf env, int val)
3720006 {
3720007     sysmsg_jmp_t msg;
3720008     msg.env = env;
3720009     msg.ret = val;
3720010     sys (SYS_LONGJMP, &msg, sizeof msg);
3720011 }

```

95.16.2 lib/setjmp/setjmp.s

« Si veda la sezione 87.49.

```

3730001 .global setjmp
3730002 .extern sys
3730003 #-----
3730004 .text
3730005 #-----
3730006 .align 4
3730007 setjmp:
3730008     #
3730009     # Previous pushes:
3730010     #

```

```

3730011 # push &env
3730012 # push back_address # made by a call to
3730013 # # setjmp() function
3730014 #
3730015 enter $8, $0
3730016 #
3730017 # sysmsg_jmp_t msg;
3730018 #
3730019 movl $0, -4(%ebp) # msg.ret = 0;
3730020 #
3730021 movl 8(%ebp), %eax # msg.env = env;
3730022 movl %eax,-8(%ebp)
3730023 #
3730024 # sys (SYS_SETJMP, &msg, sizeof msg);
3730025 #
3730026 lea -8(%ebp), %eax
3730027 pushl $8 # sizeof msg
3730028 pushl %eax # &msg
3730029 pushl $47 # SYS_SETJMP
3730030 call sys
3730031 add $4, %esp
3730032 add $4, %esp
3730033 add $4, %esp
3730034 #
3730035 # return (msg.ret);
3730036 #
3730037 movl -4(%ebp), %eax
3730038 leave
3730039 ret

```

## 95.17 os32: «lib/signal.h»

Si veda la sezione 91.3.

```

3740001 #ifndef _SIGNAL_H
3740002 #define _SIGNAL_H 1
3740003 //-----
3740004 #include <sys/types.h>
3740005 //-----
3740006 #define SIGHUP 1
3740007 #define SIGINT 2
3740008 #define SIGQUIT 3
3740009 #define SIGILL 4
3740010 #define SIGABRT 6
3740011 #define SIGFPE 8
3740012 #define SIGKILL 9
3740013 #define SIGSEGV 11
3740014 #define SIGPIPE 13
3740015 #define SIGALRM 14
3740016 #define SIGTERM 15
3740017 #define SIGSTOP 17
3740018 #define SIGTSTP 18
3740019 #define SIGCONT 19
3740020 #define SIGCHLD 20
3740021 #define SIGTTIN 21
3740022 #define SIGTTOU 22
3740023 #define SIGUSR1 30
3740024 #define SIGUSR2 31
3740025 //-----
3740026 typedef int sig_atomic_t;
3740027 typedef void (*sighandler_t) (int); // [1]
3740028 //
3740029 // [1] The type 'sighandler_t' is a pointer to a
3740030 // function for the signal handling, with a parameter
3740031 // of type 'int', returning 'void'.
3740032 //
3740033 //-----
3740034 // Special function used to call the real signal
3740035 // handler. This function will return to the 'back'
3740036 // address, instead where it was called.
3740037 //
3740038 void _sighandler_wrapper (uint32_t handler,
3740039 uint32_t signal, uint32_t back);
3740040 //-----
3740041 // Special undeclarable functions.
3740042 //
3740043 #define SIG_ERR ((sighandler_t) -1) // [2]
3740044 #define SIG_DFL ((sighandler_t) 0) // [2]
3740045 #define SIG_IGN ((sighandler_t) 1) // [2]
3740046 //
3740047 // [2] It transforms an integer number into a
3740048 // 'sighandler_t' type, that is, a pointer
3740049 // to a function that does not exists really.
3740050 //
3740051 //-----
3740052 sighandler_t signal (int sig, sighandler_t handler);
3740053 int kill (pid_t pid, int sig);

```

```

3740054 int raise (int sig);
3740055 //-----
3740056 #endif

```

95.17.1 lib/signal/\_sighandler\_wrapper.s ..... 793

95.17.2 lib/signal/kill.c ..... 794

95.17.3 lib/signal/signal.c ..... 794

95.17.1 lib/signal/\_sighandler\_wrapper.s

Si veda la sezione 87.52.

```

3750001 .global _sighandler_wrapper
3750002 #-----
3750003 .section .text
3750004 #-----
3750005 # Port input byte.
3750006 #-----
3750007 _sighandler_wrapper:
3750008 #
3750009 # Current stack is:
3750010 #
3750011 # push %eip # Back from interrupted code.
3750012 # push <sig_num> # Signal number.
3750013 # push <sig_handler> # Signal handler address
3750014 #
3750015 # Please note that THERE IS NO RETURN ADDRESS!
3750016 # Instead you find the signal handler address
3750017 # there.
3750018 #
3750019 # This routine should have to call the signal
3750020 # handler function, and then return back to the
3750021 # interrupted code.
3750022 #
3750023 enter $0, $0 # No local variables.
3750024 pushf
3750025 pusha
3750026 .equ SIG_HAND, 4 # First argument. [1]
3750027 .equ SIG_NUM, 8 # Second argument. [1]
3750028 #
3750029 # [1] This function is called without the return
3750030 # address inside the stack. So the arguments
3750031 # are 4 bytes nearer than the usual.
3750032 #
3750033 mov SIG_NUM(%ebp), %edx # Copy the signal
3750034 # number into EDX.
3750035 mov SIG_HAND(%ebp), %eax # Copy the signal
3750036 # handler function
3750037 # address into EAX.
3750038 push %edx # Prepare argument for
3750039 # the signal
3750040 # handler function.
3750041 call *%eax # Call the signal
3750042 # handler function.
3750043 add $4, %esp # Pop the signal
3750044 # number argument.
3750045 popa
3750046 popf
3750047 leave
3750048 #
3750049 # Now we are back to the same stack as the
3750050 # beginning:
3750051 #
3750052 # push %eip # back from interrupted code.
3750053 # push <sig_num>
3750054 # push <sig_handler>
3750055 # push %eip # back from
3750056 # # _sighandler_wrapper()
3750057 #
3750058 # The stack pointer must be modified before
3750059 # returning, so that the address to the original
3750060 # interrupted instruction is used for return.
3750061 # Without such modification, the RET
3750062 # instruction would find the signal handler address
3750063 # instead!
3750064 #
3750065 add $4, %esp
3750066 add $4, %esp
3750067 #
3750068 # Now we are ready to return to the original
3750069 # interrupted address!
3750070 #
3750071 ret
3750072

```

## 95.17.2 lib/signal/kill.c

« Si veda la sezione 87.29.

```

3760001 #include <sys/os32.h>
3760002 #include <sys/types.h>
3760003 #include <signal.h>
3760004 #include <errno.h>
3760005 #include <string.h>
3760006 //-----
3760007 int
3760008 kill (pid_t pid, int sig)
3760009 {
3760010     sysmsg_kill_t msg;
3760011     if (pid < -1) // Currently unsupported.
3760012     {
3760013         errset (ESRCH);
3760014         return (-1);
3760015     }
3760016     msg.pid = pid;
3760017     msg.signal = sig;
3760018     msg.ret = 0;
3760019     msg.errno = 0;
3760020     sys (SYS_KILL, &msg, (sizeof msg));
3760021     errno = msg.errno;
3760022     errln = msg.errln;
3760023     strncpy (errfn, msg.errfn, PATH_MAX);
3760024     return (msg.ret);
3760025 }

```

## 95.17.3 lib/signal/signal.c

« Si veda la sezione 87.52.

```

3770001 #include <sys/os32.h>
3770002 #include <sys/types.h>
3770003 #include <signal.h>
3770004 #include <errno.h>
3770005 #include <string.h>
3770006 //-----
3770007 sighandler_t
3770008 signal (int sig, sighandler_t handler)
3770009 {
3770010     sysmsg_signal_t msg;
3770011
3770012     msg.signal = sig;
3770013     msg.handler = handler;
3770014     msg.wrapper = (uintptr_t) _sighandler_wrapper;
3770015     msg.ret = SIG_DFL;
3770016     msg.errno = 0;
3770017     sys (SYS_SIGNAL, &msg, (sizeof msg));
3770018     errno = msg.errno;
3770019     errln = msg.errln;
3770020     strncpy (errfn, msg.errfn, PATH_MAX);
3770021     return (msg.ret);
3770022 }

```

## 95.18 os32: «lib/stdio.h»

« Si veda la sezione 88.112.

```

3780001 #ifndef _STDIO_H
3780002 #define _STDIO_H      1
3780003 //-----
3780004 #include <restrict.h>
3780005 #include <stdarg.h>
3780006 #include <stdint.h>
3780007 #include <limits.h>
3780008 #include <NULL.h>
3780009 #include <size_t.h>
3780010 #include <sys/types.h>
3780011 #include <SEEK.h> // SEEK_CUR, SEEK_SET,
3780012 //                // SEEK_END
3780013 //-----
3780014 #define BUFSIZ      8192 // At least the
3780015 //                        // file
3780016 //                        // system max zone
3780017 //                        // size.
3780018 #define _IOFBF      0 // Input-output
3780019 //                        // fully
3780020 //                        // buffered.
3780021 #define _IOLBF      1 // Input-output
3780022 //                        // line
3780023 //                        // buffered.
3780024 #define _IONBF      2 // Input-output
3780025 //                        // with
3780026 //                        // no buffering.
3780027

```

```

3780028 #define L_tmpnam     FILENAME_MAX // <limits.h>
3780029
3780030 #define FOPEN_MAX    OPEN_MAX // <limits.h>
3780031 #define FILENAME_MAX NAME_MAX // <limits.h>
3780032 #define TMP_MAX      0x7FFF
3780033
3780034 #define EOF          (-1) // Must be a
3780035 //                        // negative
3780036 //                        // value.
3780037 //-----
3780038 typedef off_t fpos_t; // 'off_t' defined in
3780039 // <sys/types.h>.
3780040
3780041 typedef struct
3780042 {
3780043     int fdn; // File descriptor number.
3780044     char error; // Error indicator.
3780045     char eof; // End of file indicator.
3780046 } FILE;
3780047
3780048 extern FILE _stream[]; // Defined inside
3780049 // 'lib/stdio/FILE.c'.
3780050
3780051 #define stdin (&_stream[0])
3780052 #define stdout (&_stream[1])
3780053 #define stderr (&_stream[2])
3780054 //-----
3780055 void clearerr (FILE * fp);
3780056 int fclose (FILE * fp);
3780057 int feof (FILE * fp);
3780058 int ferror (FILE * fp);
3780059 int fflush (FILE * fp);
3780060 int fgetc (FILE * fp);
3780061 int fgetpos (FILE * restrict fp, fpos_t * restrict pos);
3780062 char *fgets (char *restrict string, int n,
3780063             FILE * restrict fp);
3780064 int fileno (FILE * fp);
3780065 FILE *fopen (const char *path, const char *mode);
3780066 int fprintf (FILE * fp, char *restrict format, ...);
3780067 int fputc (int c, FILE * fp);
3780068 int fputs (const char *restrict string, FILE * restrict fp);
3780069 size_t fread (void *restrict buffer, size_t size,
3780070             size_t nmemb, FILE * restrict fp);
3780071 FILE *freopen (const char *restrict path,
3780072              const char *restrict mode,
3780073              FILE * restrict fp);
3780074 int fscanf (FILE * restrict fp,
3780075            const char *restrict format, ...);
3780076 int fseek (FILE * fp, long int offset, int whence);
3780077 int fsetpos (FILE * fp, fpos_t * pos);
3780078 long int ftell (FILE * fp);
3780079 off_t ftello (FILE * fp);
3780080 size_t fwrite (const void *restrict buffer,
3780081             size_t size, size_t nmemb,
3780082             FILE * restrict fp);
3780083 #define getc(p) (fgetc (p))
3780084 int getchar (void);
3780085 char *gets (char *string);
3780086 void perror (const char *string);
3780087 int printf (const char *restrict format, ...);
3780088 #define putc(c, p) (fputc ((c), (p)))
3780089 int putchar (int c);
3780090 int puts (const char *string);
3780091 void rewind (FILE * fp);
3780092 int scanf (const char *restrict format, ...);
3780093 void setbuf (FILE * restrict fp, char *restrict buffer);
3780094 int setvbuf (FILE * restrict fp, char *restrict buffer,
3780095            int buf_mode, size_t size);
3780096 int snprintf (char *restrict string, size_t size,
3780097            const char *restrict format, ...);
3780098 int sprintf (char *restrict string,
3780099            const char *restrict format, ...);
3780100 int sscanf (char *restrict string,
3780101            const char *restrict format, ...);
3780102 int vfprintf (FILE * fp, char *restrict format,
3780103            va_list arg);
3780104 int vfscanf (FILE * restrict fp,
3780105            const char *restrict format, va_list arg);
3780106 int vprintf (const char *restrict format, va_list arg);
3780107 int vscanf (const char *restrict format, va_list ap);
3780108 int vsnprintf (char *restrict string, size_t size,
3780109            const char *restrict format, va_list arg);
3780110 int vsprintf (char *restrict string,
3780111            const char *restrict format, va_list arg);
3780112 int vsscanf (const char *string, const char *format,
3780113            va_list ap);
3780114 //-----

```



3790115	#endif	
95.18.1	lib/stdio/FILE.c	796
95.18.2	lib/stdio/clearerr.c	797
95.18.3	lib/stdio/fclose.c	797
95.18.4	lib/stdio/feof.c	797
95.18.5	lib/stdio/ferror.c	797
95.18.6	lib/stdio/fflush.c	798
95.18.7	lib/stdio/fgetc.c	798
95.18.8	lib/stdio/fgetpos.c	798
95.18.9	lib/stdio/fgets.c	798
95.18.10	lib/stdio/fileno.c	799
95.18.11	lib/stdio/fopen.c	799
95.18.12	lib/stdio/fprintf.c	800
95.18.13	lib/stdio/fputc.c	800
95.18.14	lib/stdio/fputs.c	801
95.18.15	lib/stdio/fread.c	801
95.18.16	lib/stdio/freopen.c	801
95.18.17	lib/stdio/fscanf.c	802
95.18.18	lib/stdio/fseek.c	802
95.18.19	lib/stdio/fseeko.c	802
95.18.20	lib/stdio/fsetpos.c	802
95.18.21	lib/stdio/ftell.c	803
95.18.22	lib/stdio/ftello.c	803
95.18.23	lib/stdio/fwrite.c	803
95.18.24	lib/stdio/getchar.c	803
95.18.25	lib/stdio/gets.c	804
95.18.26	lib/stdio/perror.c	804
95.18.27	lib/stdio/printf.c	805
95.18.28	lib/stdio/putchar.c	805
95.18.29	lib/stdio/puts.c	805
95.18.30	lib/stdio/rewind.c	806
95.18.31	lib/stdio/scanf.c	806
95.18.32	lib/stdio/setbuf.c	806
95.18.33	lib/stdio/setvbuf.c	806
95.18.34	lib/stdio/snprintf.c	806
95.18.35	lib/stdio/sprintf.c	806
95.18.36	lib/stdio/sscanf.c	807
95.18.37	lib/stdio/vfprintf.c	807
95.18.38	lib/stdio/vfscanf.c	807
95.18.39	lib/stdio/vfsscanf.c	807
95.18.40	lib/stdio/vprintf.c	827
95.18.41	lib/stdio/vscanf.c	827
95.18.42	lib/stdio/vsnprintf.c	828
95.18.43	lib/stdio/vsprintf.c	843
95.18.44	lib/stdio/vsscanf.c	844

95.18.1 lib/stdio/FILE.c

«

Si veda la sezione 91.3.

```

3790001 #include <stdio.h>
3790002 //
3790003 // There must be room for at least 'FOPEN_MAX'
3790004 // elements.
3790005 //
3790006 FILE _stream[FOPEN_MAX];
3790007 //-----
3790008 void
3790009 _stdio_stream_setup (void)
3790010 {
3790011     _stream[0].fdn = 0;
3790012     _stream[0].error = 0;
3790013     _stream[0].eof = 0;
3790014
3790015     _stream[1].fdn = 1;
3790016     _stream[1].error = 0;
3790017     _stream[1].eof = 0;
3790018
3790019     _stream[2].fdn = 2;
3790020     _stream[2].error = 0;
3790021     _stream[2].eof = 0;
3790022 }

```

95.18.2 lib/stdio/clearerr.c

«

Si veda la sezione 88.12.

```

3800001 #include <stdio.h>
3800002 //-----
3800003 void
3800004 clearerr (FILE * fp)
3800005 {
3800006     if (fp != NULL)
3800007     {
3800008         fp->error = 0;
3800009         fp->eof = 0;
3800010     }
3800011 }

```

95.18.3 lib/stdio/fclose.c

«

Si veda la sezione 88.28.

```

3810001 #include <stdio.h>
3810002 #include <unistd.h>
3810003 //-----
3810004 int
3810005 fclose (FILE * fp)
3810006 {
3810007     return (close (fp->fdn));
3810008 }

```

95.18.4 lib/stdio/feof.c

«

Si veda la sezione 88.29.

```

3820001 #include <stdio.h>
3820002 //-----
3820003 int
3820004 feof (FILE * fp)
3820005 {
3820006     if (fp != NULL)
3820007     {
3820008         return (fp->eof);
3820009     }
3820010     return (0);
3820011 }

```

95.18.5 lib/stdio/ferror.c

«

Si veda la sezione 88.30.

```

3830001 #include <stdio.h>
3830002 //-----
3830003 int
3830004 ferror (FILE * fp)
3830005 {
3830006     if (fp != NULL)
3830007     {
3830008         return (fp->error);
3830009     }
3830010     return (0);
3830011 }

```

## 95.18.6 lib/stdio/fflush.c

&lt;&lt;

Si veda la sezione 88.31.

```

3840001 #include <stdio.h>
3840002 //-----
3840003 int
3840004 fflush (FILE * fp)
3840005 {
3840006     //
3840007     // The os32 library does not have any buffered data.
3840008     //
3840009     return (0);
3840010 }

```

## 95.18.7 lib/stdio/fgetc.c

&lt;&lt;

Si veda la sezione 88.32.

```

3850001 #include <stdio.h>
3850002 #include <sys/types.h>
3850003 #include <unistd.h>
3850004 //-----
3850005 int
3850006 fgetc (FILE * fp)
3850007 {
3850008     ssize_t size_read;
3850009     int c; // Character read.
3850010     //
3850011     for (c = 0;;)
3850012     {
3850013         size_read = read (fp->fdn, &c, (size_t) 1);
3850014         //
3850015         if (size_read <= 0)
3850016         {
3850017             //
3850018             // It is the end of file (zero) otherwise
3850019             // there is a
3850020             // problem (a negative value): return 'EOF'.
3850021             //
3850022             return (EOF);
3850023         }
3850024         //
3850025         // Valid read: end of scan.
3850026         //
3850027         return (c);
3850028     }
3850029 }

```

## 95.18.8 lib/stdio/fgetpos.c

&lt;&lt;

Si veda la sezione 88.33.

```

3860001 #include <stdio.h>
3860002 //-----
3860003 int
3860004 fgetpos (FILE * restrict fp, fpos_t * restrict pos)
3860005 {
3860006     long int position;
3860007     //
3860008     if (fp != NULL)
3860009     {
3860010         position = ftell (fp);
3860011         if (position >= 0)
3860012         {
3860013             *pos = position;
3860014             return (0);
3860015         }
3860016     }
3860017     return (-1);
3860018 }

```

## 95.18.9 lib/stdio/fgets.c

&lt;&lt;

Si veda la sezione 88.34.

```

3870001 #include <stdio.h>
3870002 #include <sys/types.h>
3870003 #include <unistd.h>
3870004 #include <stddef.h>
3870005 //-----
3870006 char *
3870007 fgets (char *restrict string, int n, FILE * restrict fp)
3870008 {
3870009     ssize_t size_read;
3870010     int b; // Index inside the string buffer.
3870011     //
3870012     for (b = 0; b < (n - 1); b++, string[b] = 0)

```

```

3870013     {
3870014         size_read = read (fp->fdn, &string[b], (size_t) 1);
3870015         //
3870016         if (size_read <= 0)
3870017         {
3870018             //
3870019             // It is the end of file (zero) otherwise
3870020             // there is a
3870021             // problem (a negative value).
3870022             //
3870023             string[b] = 0;
3870024             break;
3870025         }
3870026         //
3870027         if (string[b] == '\n')
3870028         {
3870029             b++;
3870030             string[b] = 0;
3870031             break;
3870032         }
3870033     }
3870034     //
3870035     // If 'b' is zero, nothing was read and 'NULL' is
3870036     // returned.
3870037     //
3870038     if (b == 0)
3870039     {
3870040         return (NULL);
3870041     }
3870042     else
3870043     {
3870044         return (string);
3870045     }
3870046 }

```

## 95.18.10 lib/stdio/fileno.c

&lt;&lt;

Si veda la sezione 88.35.

```

3880001 #include <stdio.h>
3880002 #include <errno.h>
3880003 //-----
3880004 int
3880005 fileno (FILE * fp)
3880006 {
3880007     if (fp != NULL)
3880008     {
3880009         return (fp->fdn);
3880010     }
3880011     errset (EBADF); // Bad file descriptor.
3880012     return (-1);
3880013 }

```

## 95.18.11 lib/stdio/fopen.c

&lt;&lt;

Si veda la sezione 88.36.

```

3890001 #include <fcntl.h>
3890002 #include <stdarg.h>
3890003 #include <stddef.h>
3890004 #include <string.h>
3890005 #include <errno.h>
3890006 #include <sys/os32.h>
3890007 #include <limits.h>
3890008 #include <stdio.h>
3890009 //-----
3890010 FILE *
3890011 fopen (const char *path, const char *mode)
3890012 {
3890013     int fdn;
3890014     //
3890015     if (strcmp (mode, "r") || strcmp (mode, "rb"))
3890016     {
3890017         fdn = open (path, O_RDONLY);
3890018     }
3890019     else if (strcmp (mode, "r+") ||
3890020             strcmp (mode, "r+b") || strcmp (mode, "rb+"))
3890021     {
3890022         fdn = open (path, O_RDWR);
3890023     }
3890024     else if (strcmp (mode, "w") || strcmp (mode, "wb"))
3890025     {
3890026         fdn = open (path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
3890027     }
3890028     else if (strcmp (mode, "w+") ||
3890029             strcmp (mode, "w+b") || strcmp (mode, "wb+"))
3890030     {

```

```

389001     fdn = open (path, O_RDWR | O_CREAT | O_TRUNC, 0666);
389002     }
389003     else if (strcmp (mode, "a") || strcmp (mode, "ab"))
389004     {
389005         fdn =
389006             open (path,
389007                 O_WRONLY | O_APPEND | O_CREAT | O_TRUNC,
389008                 0666);
389009     }
389010     else if (strcmp (mode, "a+") ||
389011             strcmp (mode, "a+b") || strcmp (mode, "ab+"))
389012     {
389013         fdn =
389014             open (path,
389015                 O_RDWR | O_APPEND | O_CREAT | O_TRUNC, 0666);
389016     }
389017     else
389018     {
389019         errset (EINVAL); // Invalid argument.
389020         return (NULL);
389021     }
389022     //
389023     // Check the file descriptor returned.
389024     //
389025     if (fdn < 0)
389026     {
389027         //
389028         // The variable 'errno' is already set.
389029         //
389030         errset (errno);
389031         return (NULL);
389032     }
389033     //
389034     // A valid file descriptor is available: convert it
389035     // into a file
389036     // stream. Please note that the file descriptor
389037     // number must be
389038     // saved inside the corresponding '_stream[]' array,
389039     // because the
389040     // file pointer do not have knowledge of the
389041     // relative position
389042     // inside the array.
389043     //
389044     _stream[fdn].fdn = fdn; // Saved the file
389045     // descriptor number.
389046     //
389047     return (&_stream[fdn]); // Returned the file
389048     // stream pointer.
389049 }

```

## 95.18.12 lib/stdio/fprintf.c

Si veda la sezione 88.91.

```

390001 #include <stdio.h>
390002 //-----
390003 int
390004 fprintf (FILE * fp, char *restrict format, ...)
390005 {
390006     va_list ap;
390007     va_start (ap, format);
390008     return (vfprintf (fp, format, ap));
390009 }

```

## 95.18.13 lib/stdio/fputc.c

Si veda la sezione 88.38.

```

391001 #include <stdio.h>
391002 #include <sys/types.h>
391003 #include <sys/os32.h>
391004 #include <string.h>
391005 #include <unistd.h>
391006 //-----
391007 int
391008 fputc (int c, FILE * fp)
391009 {
391010     ssize_t size_written;
391011     char character = (char) c;
391012     size_written = write (fp->fdn, &character, (size_t) 1);
391013     if (size_written < 0)
391014     {
391015         fp->eof = 1;
391016         return (EOF);
391017     }
391018     return (c);
391019 }

```

## 95.18.14 lib/stdio/fputs.c

Si veda la sezione 88.39.

```

392001 #include <stdio.h>
392002 #include <string.h>
392003 //-----
392004 int
392005 fputs (const char *restrict string, FILE * restrict fp)
392006 {
392007     int i; // Index inside the string to be
392008     // printed.
392009     int status;
392010
392011     for (i = 0; i < strlen (string); i++)
392012     {
392013         status = fputc (string[i], fp);
392014         if (status == EOF)
392015         {
392016             fp->eof = 1;
392017             return (EOF);
392018         }
392019     }
392020     return (0);
392021 }

```

## 95.18.15 lib/stdio/fread.c

Si veda la sezione 88.40.

```

393001 #include <unistd.h>
393002 #include <stdio.h>
393003 //-----
393004 size_t
393005 fread (void *restrict buffer, size_t size,
393006        size_t nmemb, FILE * restrict fp)
393007 {
393008     ssize_t size_read;
393009     size_read =
393010         read (fp->fdn, buffer, (size_t) (size * nmemb));
393011     if (size_read == 0)
393012     {
393013         fp->eof = 1;
393014         return ((size_t) 0);
393015     }
393016     else if (size_read < 0)
393017     {
393018         fp->error = 1;
393019         return ((size_t) 0);
393020     }
393021     else
393022     {
393023         return ((size_t) (size_read / size));
393024     }
393025 }

```

## 95.18.16 lib/stdio/freopen.c

Si veda la sezione 88.36.

```

394001 #include <fcntl.h>
394002 #include <stdarg.h>
394003 #include <stddef.h>
394004 #include <string.h>
394005 #include <errno.h>
394006 #include <sys/os32.h>
394007 #include <limits.h>
394008 #include <stdio.h>
394009 //-----
394010 FILE *
394011 freopen (const char *restrict path,
394012         const char *restrict mode, FILE * restrict fp)
394013 {
394014     int status;
394015     FILE *fp_new;
394016     //
394017     if (fp == NULL)
394018     {
394019         return (NULL);
394020     }
394021     //
394022     status = fclose (fp);
394023     if (status != 0)
394024     {
394025         fp->error = 1;
394026         return (NULL);
394027     }
394028     //

```

```

3940029     fp_new = fopen (path, mode);
3940030     //
3940031     if (fp_new == NULL)
3940032     {
3940033         return (NULL);
3940034     }
3940035     //
3940036     if (fp_new != fp)
3940037     {
3940038         fclose (fp_new);
3940039         return (NULL);
3940040     }
3940041     //
3940042     return (fp_new);
3940043 }

```

## 95.18.17 lib/stdio/fscanf.c

&lt;&lt;

Si veda la sezione 88.102.

```

3950001 #include <stdio.h>
3950002 //-----
3950003 int
3950004 fscanf (FILE * restrict fp,
3950005         const char * restrict format, ...)
3950006 {
3950007     va_list ap;
3950008     va_start (ap, format);
3950009     return vfscanf (fp, format, ap);
3950010 }

```

## 95.18.18 lib/stdio/fseek.c

&lt;&lt;

Si veda la sezione 88.44.

```

3960001 #include <stdio.h>
3960002 #include <unistd.h>
3960003 //-----
3960004 int
3960005 fseek (FILE * fp, long int offset, int whence)
3960006 {
3960007     off_t off_new;
3960008     off_new = lseek (fp->fdn, (off_t) offset, whence);
3960009     if (off_new < 0)
3960010     {
3960011         fp->error = 1;
3960012         return (-1);
3960013     }
3960014     else
3960015     {
3960016         fp->eof = 0;
3960017         return (0);
3960018     }
3960019 }

```

## 95.18.19 lib/stdio/fseeko.c

&lt;&lt;

Si veda la sezione 88.44.

```

3970001 #include <stdio.h>
3970002 #include <unistd.h>
3970003 //-----
3970004 int
3970005 fseeko (FILE * fp, off_t offset, int whence)
3970006 {
3970007     off_t off_new;
3970008     off_new = lseek (fp->fdn, offset, whence);
3970009     if (off_new < 0)
3970010     {
3970011         fp->error = 1;
3970012         return (-1);
3970013     }
3970014     else
3970015     {
3970016         return (0);
3970017     }
3970018 }

```

## 95.18.20 lib/stdio/fsetpos.c

&lt;&lt;

Si veda la sezione 88.33.

```

3980001 #include <stdio.h>
3980002 //-----
3980003 int
3980004 fsetpos (FILE * fp, fpos_t * pos)

```

```

3980005 {
3980006     long int position;
3980007     //
3980008     if (fp != NULL)
3980009     {
3980010         position = fseek (fp, (long int) *pos, SEEK_SET);
3980011         if (position >= 0)
3980012         {
3980013             *pos = position;
3980014             return (0);
3980015         }
3980016     }
3980017     return (-1);
3980018 }

```

## 95.18.21 lib/stdio/ftell.c

&lt;&lt;

Si veda la sezione 88.47.

```

3990001 #include <stdio.h>
3990002 #include <unistd.h>
3990003 //-----
3990004 long int
3990005 ftell (FILE * fp)
3990006 {
3990007     return ((long int) lseek (fp->fdn, (off_t) 0, SEEK_CUR));
3990008 }

```

## 95.18.22 lib/stdio/ftello.c

&lt;&lt;

Si veda la sezione 88.47.

```

4000001 #include <stdio.h>
4000002 #include <unistd.h>
4000003 //-----
4000004 off_t
4000005 ftello (FILE * fp)
4000006 {
4000007     return (lseek (fp->fdn, (off_t) 0, SEEK_CUR));
4000008 }

```

## 95.18.23 lib/stdio/fwrite.c

&lt;&lt;

Si veda la sezione 88.49.

```

4010001 #include <unistd.h>
4010002 #include <stdio.h>
4010003 //-----
4010004 size_t
4010005 fwrite (const void * restrict buffer, size_t size,
4010006         size_t nmemb, FILE * restrict fp)
4010007 {
4010008     ssize_t size_written;
4010009     size_written =
4010010     write (fp->fdn, buffer, (size_t) (size * nmemb));
4010011     if (size_written < 0)
4010012     {
4010013         fp->error = 1;
4010014         return ((size_t) 0);
4010015     }
4010016     else
4010017     {
4010018         return ((size_t) (size_written / size));
4010019     }
4010020 }

```

## 95.18.24 lib/stdio/getchar.c

&lt;&lt;

Si veda la sezione 88.32.

```

4020001 #include <stdio.h>
4020002 #include <sys/types.h>
4020003 #include <unistd.h>
4020004 //-----
4020005 int
4020006 getchar (void)
4020007 {
4020008     ssize_t size_read;
4020009     int c; // Character read.
4020010     //
4020011     for (c = 0;;)
4020012     {
4020013         size_read = read (STDIN_FILENO, &c, (size_t) 1);
4020014         //
4020015         if (size_read <= 0)
4020016         {

```

```

402017 //
402018 // It is the end of file (zero) otherwise
402019 // there is a
402020 // problem (a negative value): return 'EOF'.
402021 //
402022 _stream[STDIN_FILENO].eof = 1;
402023 return (EOF);
402024 }
402025 //
402026 // Valid read.
402027 //
402028 if (size_read == 0)
402029 {
402030 //
402031 // If no character is ready inside the
402032 // keyboard buffer, just
402033 // retry.
402034 //
402035 continue;
402036 }
402037 //
402038 // End of scan.
402039 //
402040 return (c);
402041 }
402042 }

```

## 95.18.25 lib/stdio/get.c

&lt;

Si veda la sezione 88.34.

```

403001 #include <stdio.h>
403002 #include <sys/types.h>
403003 #include <unistd.h>
403004 #include <stddef.h>
403005 //-----
403006 char *
403007 gets (char *string)
403008 {
403009     ssize_t size_read;
403010     int b; // Index inside the string buffer.
403011     //
403012     for (b = 0;; b++, string[b] = 0)
403013     {
403014         size_read =
403015             read (STDIN_FILENO, &string[b], (size_t) 1);
403016         //
403017         if (size_read <= 0)
403018         {
403019             //
403020             // It is the end of file (zero) otherwise
403021             // there is a
403022             // problem (a negative value).
403023             //
403024             _stream[STDIN_FILENO].eof = 1;
403025             string[b] = 0;
403026             break;
403027         }
403028         //
403029         if (string[b] == '\n')
403030         {
403031             b++;
403032             string[b] = 0;
403033             break;
403034         }
403035     }
403036     //
403037     // If 'b' is zero, nothing was read and 'NULL' is
403038     // returned.
403039     //
403040     if (b == 0)
403041     {
403042         return (NULL);
403043     }
403044     else
403045     {
403046         return (string);
403047     }
403048 }

```

## 95.18.26 lib/stdio/perror.c

&lt;

Si veda la sezione 88.90.

```

404001 #include <stdio.h>
404002 #include <errno.h>
404003 #include <stddef.h>

```

```

404004 #include <string.h>
404005 //-----
404006 void
404007 perror (const char *string)
404008 {
404009     //
404010     // If errno is zero, there is nothing to show.
404011     //
404012     if (errno == 0)
404013     {
404014         return;
404015     }
404016     //
404017     // Show the string if there is one.
404018     //
404019     if (string != NULL && strlen (string) > 0)
404020     {
404021         printf ("%s: ", string);
404022     }
404023     //
404024     // Show the translated error.
404025     //
404026     if (errfn[0] != 0 && errln != 0)
404027     {
404028         printf ("[%s:%u:%i] %s\n",
404029             errfn, errln, errno, strerror (errno));
404030     }
404031     else
404032     {
404033         printf ("[%i] %s\n", errno, strerror (errno));
404034     }
404035 }

```

## 95.18.27 lib/stdio/printf.c

&gt;

Si veda la sezione 88.91.

```

405001 #include <stdio.h>
405002 //-----
405003 int
405004 printf (const char *restrict format, ...)
405005 {
405006     va_list ap;
405007     va_start (ap, format);
405008     return (vprintf (format, ap));
405009 }

```

## 95.18.28 lib/stdio/putchar.c

&gt;

Si veda la sezione 88.38.

```

406001 #include <stdio.h>
406002 #include <sys/types.h>
406003 #include <sys/os32.h>
406004 #include <string.h>
406005 #include <unistd.h>
406006 //-----
406007 int
406008 putchar (int c)
406009 {
406010     return (fputc (c, stdout));
406011 }

```

## 95.18.29 lib/stdio/puts.c

&gt;

Si veda la sezione 88.39.

```

407001 #include <stdio.h>
407002 //-----
407003 int
407004 puts (const char *string)
407005 {
407006     int status;
407007     status = printf ("%s\n", string);
407008     if (status < 0)
407009     {
407010         return (EOF);
407011     }
407012     else
407013     {
407014         return (status);
407015     }
407016 }

```

## 95.18.30 lib/stdio/rewind.c

&lt;&lt;

Si veda la sezione 88.100.

```

408001 #include <stdio.h>
408002 //-----
408003 void
408004 rewind (FILE * fp)
408005 {
408006     (void) fseek (fp, 0L, SEEK_SET);
408007     fp->error = 0;
408008 }

```

## 95.18.31 lib/stdio/scanf.c

&lt;&lt;

Si veda la sezione 88.102.

```

409001 #include <stdio.h>
409002 //-----
409003 int
409004 scanf (const char *restrict format, ...)
409005 {
409006     va_list ap;
409007     va_start (ap, format);
409008     return vfscanf (stdin, format, ap);
409009 }

```

## 95.18.32 lib/stdio/setbuf.c

&lt;&lt;

Si veda la sezione 88.103.

```

410001 #include <stdio.h>
410002 //-----
410003 void
410004 setbuf (FILE * restrict fp, char *restrict buffer)
410005 {
410006     //
410007     // The os32 library does not have any buffered data.
410008     //
410009     return;
410010 }

```

## 95.18.33 lib/stdio/setvbuf.c

&lt;&lt;

Si veda la sezione 88.103.

```

411001 #include <stdio.h>
411002 //-----
411003 int
411004 setvbuf (FILE * restrict fp, char *restrict buffer,
411005         int buf_mode, size_t size)
411006 {
411007     //
411008     // The os32 library does not have any buffered data.
411009     //
411010     return (0);
411011 }

```

## 95.18.34 lib/stdio/snprintf.c

&lt;&lt;

Si veda la sezione 88.91.

```

412001 #include <stdio.h>
412002 #include <stdarg.h>
412003 //-----
412004 int
412005 snprintf (char *restrict string, size_t size,
412006          const char *restrict format, ...)
412007 {
412008     va_list ap;
412009     va_start (ap, format);
412010     return vsnprintf (string, size, format, ap);
412011 }

```

## 95.18.35 lib/stdio/sprintf.c

&lt;&lt;

Si veda la sezione 88.91.

```

413001 #include <stdio.h>
413002 #include <stdarg.h>
413003 //-----
413004 int
413005 sprintf (char *restrict string,
413006         const char *restrict format, ...)
413007 {
413008     va_list ap;
413009     va_start (ap, format);

```

```

413001     return vsnprintf (string, (size_t) BUFSIZ, format, ap);
413002 }

```

## 95.18.36 lib/stdio/sscanf.c

&lt;&lt;

Si veda la sezione 88.102.

```

414001 #include <stdio.h>
414002 //-----
414003 int
414004 sscanf (char *restrict string,
414005         const char *restrict format, ...)
414006 {
414007     va_list ap;
414008     va_start (ap, format);
414009     return vsscanf (string, format, ap);
414010 }

```

## 95.18.37 lib/stdio/vfprintf.c

&lt;&lt;

Si veda la sezione 88.137.

```

415001 #include <stdio.h>
415002 #include <sys/types.h>
415003 #include <sys/os32.h>
415004 #include <string.h>
415005 #include <unistd.h>
415006 //-----
415007 int
415008 vfprintf (FILE * fp, char *restrict format, va_list arg)
415009 {
415010     ssize_t size_written;
415011     size_t size;
415012     size_t size_total;
415013     int status;
415014     char string[BUFSIZ];
415015     char *buffer = string;
415016     //
415017     buffer[0] = 0;
415018     status = vsprintf (buffer, format, arg);
415019     //
415020     size = strlen (buffer);
415021     if (size >= BUFSIZ)
415022     {
415023         size = BUFSIZ;
415024     }
415025     //
415026     for (size_total = 0, size_written = 0;
415027         size_total < size;
415028         size_total += size_written, buffer += size_written)
415029     {
415030         size_written =
415031             write (fp->fdn, buffer, size - size_total);
415032         if (size_written < 0)
415033         {
415034             return (size_total);
415035         }
415036     }
415037     return (size);
415038 }

```

## 95.18.38 lib/stdio/vfscanf.c

&lt;&lt;

Si veda la sezione 88.138.

```

416001 #include <stdio.h>
416002 //-----
416003 int vfscanf (FILE * restrict fp, const char *string,
416004             const char *restrict format, va_list ap);
416005 //-----
416006 int
416007 vfscanf (FILE * restrict fp,
416008         const char *restrict format, va_list ap)
416009 {
416010     return (vfscanf (fp, NULL, format, ap));
416011 }
416012 }
416013 }
416014 //-----

```

## 95.18.39 lib/stdio/vfsscanf.c

&lt;&lt;

Si veda la sezione 88.138.

```

417001 #include <stdint.h>
417002 #include <stdbool.h>

```



```

4170177     }
4170178     }
4170179     if (format[f] == '%' && format[f + 1] == '%')
4170180     {
4170181         // ----- Matching a literal '%'.
4170182         f++;
4170183         if (format[f] == *input)
4170184         {
4170185             input++;
4170186             f++;
4170187             continue;
4170188         }
4170189         else
4170190         {
4170191             return (ass_or_eof
4170192                 (consumed, assigned));
4170193         }
4170194     }
4170195     if (format[f] == '%')
4170196     {
4170197         // ----- Percent of a specifier.
4170198         f++;
4170199         specifier = 1;
4170200         specifier_flags = 1;
4170201         continue;
4170202     }
4170203 }
4170204 //
4170205 if (specifier && specifier_flags)
4170206 {
4170207     // -----
4170208     // The context is inside
4170209     // specifier flags.
4170210     // -----
4170211     if (format[f] == '*')
4170212     {
4170213         // ---- Assignment suppression star.
4170214         flag_star = 1;
4170215         f++;
4170216     }
4170217     else
4170218     {
4170219         // -----
4170220         // End of flags and begin of
4170221         // specifier length.
4170222         // -----
4170223         specifier_flags = 0;
4170224         specifier_width = 1;
4170225     }
4170226 }
4170227 //
4170228 if (specifier && specifier_width)
4170229 {
4170230     // -----
4170231     // The context is inside a
4170232     // specifier width.
4170233     // -----
4170234     for (w = 0;
4170235          format[f] >= '0'
4170236          && format[f] <= '9'
4170237          && w < WIDTH_MAX; w++)
4170238     {
4170239         width_string[w] = format[f];
4170240         f++;
4170241     }
4170242     width_string[w] = '\0';
4170243     width = atoi (width_string);
4170244     if (width > WIDTH_MAX)
4170245     {
4170246         width = WIDTH_MAX;
4170247     }
4170248     // -----
4170249     // A zero width means an unspecified
4170250     // limit for the field
4170251     // length.
4170252     // -----
4170253     // End of spec. width and
4170254     // begin of spec. type.
4170255     // -----
4170256     specifier_width = 0;
4170257     specifier_type = 1;
4170258 }
4170259 //
4170260 if (specifier && specifier_type)
4170261 {
4170262     //

```

```

4170264     // Specifiers with length modifier.
4170265     //
4170266     if (format[f] == 'h' && format[f + 1] == 'h')
4170267     {
4170268         // ----- char.
4170269         if (format[f + 2] == 'd')
4170270         {
4170271             // ----- signed char, base 10.
4170272             value_i =
4170273                 strtointmax (input, &next, 10,
4170274                             width);
4170275             if (input == next)
4170276             {
4170277                 return (ass_or_eof
4170278                     (consumed, assigned));
4170279             }
4170280             consumed++;
4170281             if (!flag_star)
4170282             {
4170283                 ptr_schar =
4170284                     va_arg (ap, signed char *);
4170285                 *ptr_schar = value_i;
4170286                 assigned++;
4170287             }
4170288             f += 3;
4170289             input = next;
4170290         }
4170291         else if (format[f + 2] == 'i')
4170292         {
4170293             // -----
4170294             // signed char, base unknown.
4170295             // -----
4170296             value_i =
4170297                 strtointmax (input, &next, 0,
4170298                             width);
4170299             if (input == next)
4170300             {
4170301                 return (ass_or_eof
4170302                     (consumed, assigned));
4170303             }
4170304             consumed++;
4170305             if (!flag_star)
4170306             {
4170307                 ptr_schar =
4170308                     va_arg (ap, signed char *);
4170309                 *ptr_schar = value_i;
4170310                 assigned++;
4170311             }
4170312             f += 3;
4170313             input = next;
4170314         }
4170315         else if (format[f + 2] == 'o')
4170316         {
4170317             // -----
4170318             // signed char, base 8.
4170319             // -----
4170320             value_i =
4170321                 strtointmax (input, &next, 8,
4170322                             width);
4170323             if (input == next)
4170324             {
4170325                 return (ass_or_eof
4170326                     (consumed, assigned));
4170327             }
4170328             consumed++;
4170329             if (!flag_star)
4170330             {
4170331                 ptr_schar =
4170332                     va_arg (ap, signed char *);
4170333                 *ptr_schar = value_i;
4170334                 assigned++;
4170335             }
4170336             f += 3;
4170337             input = next;
4170338         }
4170339         else if (format[f + 2] == 'u')
4170340         {
4170341             // -----
4170342             // unsigned char, base 10.
4170343             // -----
4170344             value_u =
4170345                 strtointmax (input, &next, 10,
4170346                             width);
4170347             if (input == next)
4170348             {
4170349                 return (ass_or_eof
4170350                     (consumed, assigned));

```



```

4170351     }
4170352     consumed++;
4170353     if (!flag_star)
4170354     {
4170355         ptr_uchar =
4170356             va_arg (ap, unsigned char *);
4170357         *ptr_uchar = value_u;
4170358         assigned++;
4170359     }
4170360     f += 3;
4170361     input = next;
4170362 }
4170363 else if (format[f + 2] == 'x'
4170364         || format[f + 2] == 'X')
4170365 {
4170366     // -----
4170367     // signed char, base 16.
4170368     // -----
4170369     value_i =
4170370         strtointmax (input, &next, 16,
4170371                     width);
4170372     if (input == next)
4170373     {
4170374         return (ass_or_eof
4170375                 (consumed, assigned));
4170376     }
4170377     consumed++;
4170378     if (!flag_star)
4170379     {
4170380         ptr_schar =
4170381             va_arg (ap, signed char *);
4170382         *ptr_schar = value_i;
4170383         assigned++;
4170384     }
4170385     f += 3;
4170386     input = next;
4170387 }
4170388 else if (format[f + 2] == 'n')
4170389 {
4170390     // -----
4170391     // signed char,
4170392     // string index counter.
4170393     // -----
4170394     ptr_schar =
4170395         va_arg (ap, signed char *);
4170396     *ptr_schar =
4170397         (signed char) (input - start +
4170398                       scanned);
4170399     f += 3;
4170400 }
4170401 else
4170402 {
4170403     // -----
4170404     // unsupported or
4170405     // unknown specifier.
4170406     // -----
4170407     f += 2;
4170408 }
4170409 }
4170410 else if (format[f] == 'h')
4170411 {
4170412     // ----- short.
4170413     if (format[f + 1] == 'd')
4170414     {
4170415         // -----
4170416         // signed short, base 10.
4170417         // -----
4170418         value_i =
4170419             strtointmax (input, &next, 10,
4170420                         width);
4170421         if (input == next)
4170422         {
4170423             return (ass_or_eof
4170424                     (consumed, assigned));
4170425         }
4170426         consumed++;
4170427         if (!flag_star)
4170428         {
4170429             ptr_sshort =
4170430                 va_arg (ap, signed short *);
4170431             *ptr_sshort = value_i;
4170432             assigned++;
4170433         }
4170434         f += 2;
4170435         input = next;
4170436     }
4170437     else if (format[f + 1] == 'i')

```

```

4170438     {
4170439         // -----
4170440         // signed
4170441         // short, base unknown.
4170442         // -----
4170443         value_i =
4170444             strtointmax (input, &next, 0,
4170445                         width);
4170446         if (input == next)
4170447         {
4170448             return (ass_or_eof
4170449                     (consumed, assigned));
4170450         }
4170451         consumed++;
4170452         if (!flag_star)
4170453         {
4170454             ptr_sshort =
4170455                 va_arg (ap, signed short *);
4170456             *ptr_sshort = value_i;
4170457             assigned++;
4170458         }
4170459         f += 2;
4170460         input = next;
4170461     }
4170462     else if (format[f + 1] == 'o')
4170463     {
4170464         // -----
4170465         // signed short, base 8.
4170466         // -----
4170467         value_i =
4170468             strtointmax (input, &next, 8,
4170469                         width);
4170470         if (input == next)
4170471         {
4170472             return (ass_or_eof
4170473                     (consumed, assigned));
4170474         }
4170475         consumed++;
4170476         if (!flag_star)
4170477         {
4170478             ptr_sshort =
4170479                 va_arg (ap, signed short *);
4170480             *ptr_sshort = value_i;
4170481             assigned++;
4170482         }
4170483         f += 2;
4170484         input = next;
4170485     }
4170486     else if (format[f + 1] == 'u')
4170487     {
4170488         // -----
4170489         // unsigned short, base 10.
4170490         // -----
4170491         value_u =
4170492             strtointmax (input, &next, 10,
4170493                         width);
4170494         if (input == next)
4170495         {
4170496             return (ass_or_eof
4170497                     (consumed, assigned));
4170498         }
4170499         consumed++;
4170500         if (!flag_star)
4170501         {
4170502             ptr_ushort =
4170503                 va_arg (ap, unsigned short *);
4170504             *ptr_ushort = value_u;
4170505             assigned++;
4170506         }
4170507         f += 2;
4170508         input = next;
4170509     }
4170510     else if (format[f + 1] == 'x'
4170511             || format[f + 2] == 'X')
4170512     {
4170513         // -----
4170514         // signed short, base 16.
4170515         // -----
4170516         value_i =
4170517             strtointmax (input, &next, 16,
4170518                         width);
4170519         if (input == next)
4170520         {
4170521             return (ass_or_eof
4170522                     (consumed, assigned));
4170523         }
4170524         consumed++;

```

```

4170525         if (!flag_star)
4170526         {
4170527             ptr_sshort =
4170528                 va_arg (ap, signed short *);
4170529             *ptr_sshort = value_i;
4170530             assigned++;
4170531         }
4170532         f += 2;
4170533         input = next;
4170534     }
4170535     else if (format[f + 1] == 'n')
4170536     {
4170537         // -----
4170538         // signed char,
4170539         // string index counter.
4170540         // -----
4170541         ptr_sshort =
4170542             va_arg (ap, signed short *);
4170543         *ptr_sshort =
4170544             (signed short) (input - start +
4170545                             scanned);
4170546
4170547         f += 2;
4170548     }
4170549     else
4170550     {
4170551         // -----
4170552         // unsupported or
4170553         // unknown specifier.
4170554         // -----
4170555         f += 1;
4170556     }
4170557     // ----- There is no 'long long int'.
4170558     else if (format[f] == 'l')
4170559     {
4170560         // ----- long int.
4170561         if (format[f + 1] == 'd')
4170562         {
4170563             // -----
4170564             // signed long, base 10.
4170565             // -----
4170566             value_i =
4170567                 strtointmax (input, &next, 10,
4170568                             width);
4170569             if (input == next)
4170570             {
4170571                 return (ass_or_eof
4170572                         (consumed, assigned));
4170573             }
4170574             consumed++;
4170575             if (!flag_star)
4170576             {
4170577                 ptr_slong =
4170578                     va_arg (ap, signed long *);
4170579                 *ptr_slong = value_i;
4170580                 assigned++;
4170581             }
4170582             f += 2;
4170583             input = next;
4170584         }
4170585         else if (format[f + 1] == 'i')
4170586         {
4170587             // -----
4170588             // signed
4170589             // long, base unknown.
4170590             // -----
4170591             value_i =
4170592                 strtointmax (input, &next, 0,
4170593                             width);
4170594             if (input == next)
4170595             {
4170596                 return (ass_or_eof
4170597                         (consumed, assigned));
4170598             }
4170599             consumed++;
4170600             if (!flag_star)
4170601             {
4170602                 ptr_slong =
4170603                     va_arg (ap, signed long *);
4170604                 *ptr_slong = value_i;
4170605                 assigned++;
4170606             }
4170607             f += 2;
4170608             input = next;
4170609         }
4170610     }
4170611     else if (format[f + 1] == 'o')
4170612     {

```

```

4170612         // -----
4170613         // signed long, base 8.
4170614         // -----
4170615         value_i =
4170616             strtointmax (input, &next, 8,
4170617                         width);
4170618         if (input == next)
4170619         {
4170620             return (ass_or_eof
4170621                     (consumed, assigned));
4170622         }
4170623         consumed++;
4170624         if (!flag_star)
4170625         {
4170626             ptr_slong =
4170627                 va_arg (ap, signed long *);
4170628             *ptr_slong = value_i;
4170629             assigned++;
4170630         }
4170631         f += 2;
4170632         input = next;
4170633     }
4170634     else if (format[f + 1] == 'u')
4170635     {
4170636         // -----
4170637         // unsigned long, base 10.
4170638         // -----
4170639         value_u =
4170640             strtointmax (input, &next, 10,
4170641                         width);
4170642         if (input == next)
4170643         {
4170644             return (ass_or_eof
4170645                     (consumed, assigned));
4170646         }
4170647         consumed++;
4170648         if (!flag_star)
4170649         {
4170650             ptr_ulong =
4170651                 va_arg (ap, unsigned long *);
4170652             *ptr_ulong = value_u;
4170653             assigned++;
4170654         }
4170655         f += 2;
4170656         input = next;
4170657     }
4170658     else if (format[f + 1] == 'x'
4170659             || format[f + 2] == 'X')
4170660     {
4170661         // -----
4170662         // signed long, base 16.
4170663         // -----
4170664         value_i =
4170665             strtointmax (input, &next, 16,
4170666                         width);
4170667         if (input == next)
4170668         {
4170669             return (ass_or_eof
4170670                     (consumed, assigned));
4170671         }
4170672         consumed++;
4170673         if (!flag_star)
4170674         {
4170675             ptr_slong =
4170676                 va_arg (ap, signed long *);
4170677             *ptr_slong = value_i;
4170678             assigned++;
4170679         }
4170680         f += 2;
4170681         input = next;
4170682     }
4170683     else if (format[f + 1] == 'n')
4170684     {
4170685         // -----
4170686         // signed char,
4170687         // string index counter.
4170688         // -----
4170689         ptr_slong =
4170690             va_arg (ap, signed long *);
4170691         *ptr_slong =
4170692             (signed long) (input - start +
4170693                             scanned);
4170694         f += 2;
4170695     }
4170696     else
4170697     {
4170698         // -----

```

```

417099 // unsupported or
417100 // unknown specifier.
417101 // -----
417102 f += 1;
417103 }
417104 }
417105 else if (format[f] == 'j')
417106 {
417107 // ----- intmax_t.
417108 if (format[f + 1] == 'd')
417109 {
417110 // ----- intmax_t, base 10.
417111 value_i =
417112 strtointmax (input, &next, 10,
417113 width);
417114 if (input == next)
417115 {
417116 return (ass_or_eof
417117 (consumed, assigned));
417118 }
417119 consumed++;
417120 if (!flag_star)
417121 {
417122 ptr_simax =
417123 va_arg (ap, intmax_t *);
417124 *ptr_simax = value_i;
417125 assigned++;
417126 }
417127 f += 2;
417128 input = next;
417129 }
417130 else if (format[f + 1] == 'i')
417131 {
417132 // -----
417133 // intmax_t, base unknown.
417134 // -----
417135 value_i =
417136 strtointmax (input, &next, 0,
417137 width);
417138 if (input == next)
417139 {
417140 return (ass_or_eof
417141 (consumed, assigned));
417142 }
417143 consumed++;
417144 if (!flag_star)
417145 {
417146 ptr_simax =
417147 va_arg (ap, intmax_t *);
417148 *ptr_simax = value_i;
417149 assigned++;
417150 }
417151 f += 2;
417152 input = next;
417153 }
417154 else if (format[f + 1] == 'o')
417155 {
417156 // -----
417157 // intmax_t, base 8.
417158 // -----
417159 value_i =
417160 strtointmax (input, &next, 8,
417161 width);
417162 if (input == next)
417163 {
417164 return (ass_or_eof
417165 (consumed, assigned));
417166 }
417167 consumed++;
417168 if (!flag_star)
417169 {
417170 ptr_simax =
417171 va_arg (ap, intmax_t *);
417172 *ptr_simax = value_i;
417173 assigned++;
417174 }
417175 f += 2;
417176 input = next;
417177 }
417178 else if (format[f + 1] == 'u')
417179 {
417180 // -----
417181 // uintmax_t, base 10.
417182 // -----
417183 value_u =
417184 strtointmax (input, &next, 10,
417185 width);

```

```

417086 if (input == next)
417087 {
417088 return (ass_or_eof
417089 (consumed, assigned));
417090 }
417091 consumed++;
417092 if (!flag_star)
417093 {
417094 ptr_uimax =
417095 va_arg (ap, uintmax_t *);
417096 *ptr_uimax = value_u;
417097 assigned++;
417098 }
417099 f += 2;
417100 input = next;
417101 }
417102 else if (format[f + 1] == 'x'
417103 || format[f + 2] == 'X')
417104 {
417105 // -----
417106 // intmax_t, base 16.
417107 // -----
417108 value_i =
417109 strtointmax (input, &next, 16,
417110 width);
417111 if (input == next)
417112 {
417113 return (ass_or_eof
417114 (consumed, assigned));
417115 }
417116 consumed++;
417117 if (!flag_star)
417118 {
417119 ptr_simax =
417120 va_arg (ap, intmax_t *);
417121 *ptr_simax = value_i;
417122 assigned++;
417123 }
417124 f += 2;
417125 input = next;
417126 }
417127 else if (format[f + 1] == 'n')
417128 {
417129 // -----
417130 // signed char,
417131 // string index counter.
417132 // -----
417133 ptr_simax = va_arg (ap, intmax_t *);
417134 *ptr_simax =
417135 (intmax_t) (input - start +
417136 scanned);
417137 f += 2;
417138 }
417139 else
417140 {
417141 // -----
417142 // unsupported or
417143 // unknown specifier.
417144 // -----
417145 f += 1;
417146 }
417147 }
417148 else if (format[f] == 'z')
417149 {
417150 // ----- size_t.
417151 if (format[f + 1] == 'd')
417152 {
417153 // -----
417154 // size_t, base 10.
417155 // -----
417156 value_i =
417157 strtointmax (input, &next, 10,
417158 width);
417159 if (input == next)
417160 {
417161 return (ass_or_eof
417162 (consumed, assigned));
417163 }
417164 consumed++;
417165 if (!flag_star)
417166 {
417167 ptr_size = va_arg (ap, size_t *);
417168 *ptr_size = value_i;
417169 assigned++;
417170 }
417171 f += 2;
417172 input = next;

```

```

4170873     }
4170874     else if (format[f + 1] == 'i')
4170875     {
4170876         // -----
4170877         // size_t, base unknown.
4170878         // -----
4170879         value_i =
4170880             strtointmax (input, &next, 0,
4170881                         width);
4170882         if (input == next)
4170883         {
4170884             return (ass_or_eof
4170885                     (consumed, assigned));
4170886         }
4170887         consumed++;
4170888         if (!flag_star)
4170889         {
4170890             ptr_size = va_arg (ap, size_t *);
4170891             *ptr_size = value_i;
4170892             assigned++;
4170893         }
4170894         f += 2;
4170895         input = next;
4170896     }
4170897     else if (format[f + 1] == 'o')
4170898     {
4170899         // -----
4170900         // size_t, base 8.
4170901         // -----
4170902         value_i =
4170903             strtointmax (input, &next, 8,
4170904                         width);
4170905         if (input == next)
4170906         {
4170907             return (ass_or_eof
4170908                     (consumed, assigned));
4170909         }
4170910         consumed++;
4170911         if (!flag_star)
4170912         {
4170913             ptr_size = va_arg (ap, size_t *);
4170914             *ptr_size = value_i;
4170915             assigned++;
4170916         }
4170917         f += 2;
4170918         input = next;
4170919     }
4170920     else if (format[f + 1] == 'u')
4170921     {
4170922         // -----
4170923         // size_t, base 10.
4170924         // -----
4170925         value_u =
4170926             strtointmax (input, &next, 10,
4170927                         width);
4170928         if (input == next)
4170929         {
4170930             return (ass_or_eof
4170931                     (consumed, assigned));
4170932         }
4170933         consumed++;
4170934         if (!flag_star)
4170935         {
4170936             ptr_size = va_arg (ap, size_t *);
4170937             *ptr_size = value_u;
4170938             assigned++;
4170939         }
4170940         f += 2;
4170941         input = next;
4170942     }
4170943     else if (format[f + 1] == 'x'
4170944             || format[f + 2] == 'X')
4170945     {
4170946         // -----
4170947         // size_t, base 16.
4170948         // -----
4170949         value_i =
4170950             strtointmax (input, &next, 16,
4170951                         width);
4170952         if (input == next)
4170953         {
4170954             return (ass_or_eof
4170955                     (consumed, assigned));
4170956         }
4170957         consumed++;
4170958         if (!flag_star)
4170959         {

```

```

4170960         ptr_size = va_arg (ap, size_t *);
4170961         *ptr_size = value_i;
4170962         assigned++;
4170963     }
4170964     f += 2;
4170965     input = next;
4170966 }
4170967 else if (format[f + 1] == 'n')
4170968 {
4170969     // -----
4170970     // signed char,
4170971     // string index counter.
4170972     // -----
4170973     ptr_size = va_arg (ap, size_t *);
4170974     *ptr_size =
4170975         (size_t) (input - start + scanned);
4170976     f += 2;
4170977 }
4170978 else
4170979 {
4170980     // -----
4170981     // unsupported or
4170982     // unknown specifier.
4170983     // -----
4170984     f += 1;
4170985 }
4170986 }
4170987 else if (format[f] == 't')
4170988 {
4170989     // ----- ptrdiff_t.
4170990     if (format[f + 1] == 'd')
4170991     {
4170992         // -----
4170993         // ptrdiff_t, base 10.
4170994         // -----
4170995         value_i =
4170996             strtointmax (input, &next, 10,
4170997                         width);
4170998         if (input == next)
4170999         {
4171000             return (ass_or_eof
4171001                     (consumed, assigned));
4171002         }
4171003         consumed++;
4171004         if (!flag_star)
4171005         {
4171006             ptr_ptrdiff =
4171007                 va_arg (ap, ptrdiff_t *);
4171008             *ptr_ptrdiff = value_i;
4171009             assigned++;
4171010         }
4171011         f += 2;
4171012         input = next;
4171013     }
4171014     else if (format[f + 1] == 'i')
4171015     {
4171016         // -----
4171017         // ptrdiff_t, base unknown.
4171018         // -----
4171019         value_i =
4171020             strtointmax (input, &next, 0,
4171021                         width);
4171022         if (input == next)
4171023         {
4171024             return (ass_or_eof
4171025                     (consumed, assigned));
4171026         }
4171027         consumed++;
4171028         if (!flag_star)
4171029         {
4171030             ptr_ptrdiff =
4171031                 va_arg (ap, ptrdiff_t *);
4171032             *ptr_ptrdiff = value_i;
4171033             assigned++;
4171034         }
4171035         f += 2;
4171036         input = next;
4171037     }
4171038     else if (format[f + 1] == 'o')
4171039     {
4171040         // -----
4171041         // ptrdiff_t, base 8.
4171042         // -----
4171043         value_i =
4171044             strtointmax (input, &next, 8,
4171045                         width);
4171046         if (input == next)

```

```

4171047     {
4171048         return (ass_or_eof
4171049             (consumed, assigned));
4171050     }
4171051     consumed++;
4171052     if (!flag_star)
4171053     {
4171054         ptr_ptrdiff =
4171055             va_arg (ap, ptrdiff_t *);
4171056         *ptr_ptrdiff = value_i;
4171057         assigned++;
4171058     }
4171059     f += 2;
4171060     input = next;
4171061 }
4171062 else if (format[f + 1] == 'u')
4171063 {
4171064     // -----
4171065     // ptrdiff_t, base 10.
4171066     // -----
4171067     value_u =
4171068         strtointmax (input, &next, 10,
4171069                     width);
4171070     if (input == next)
4171071     {
4171072         return (ass_or_eof
4171073             (consumed, assigned));
4171074     }
4171075     consumed++;
4171076     if (!flag_star)
4171077     {
4171078         ptr_ptrdiff =
4171079             va_arg (ap, ptrdiff_t *);
4171080         *ptr_ptrdiff = value_u;
4171081         assigned++;
4171082     }
4171083     f += 2;
4171084     input = next;
4171085 }
4171086 else if (format[f + 1] == 'x'
4171087         || format[f + 2] == 'X')
4171088 {
4171089     // -----
4171090     // ptrdiff_t, base 16.
4171091     // -----
4171092     value_i =
4171093         strtointmax (input, &next, 16,
4171094                     width);
4171095     if (input == next)
4171096     {
4171097         return (ass_or_eof
4171098             (consumed, assigned));
4171099     }
4171100     consumed++;
4171101     if (!flag_star)
4171102     {
4171103         ptr_ptrdiff =
4171104             va_arg (ap, ptrdiff_t *);
4171105         *ptr_ptrdiff = value_i;
4171106         assigned++;
4171107     }
4171108     f += 2;
4171109     input = next;
4171110 }
4171111 else if (format[f + 1] == 'n')
4171112 {
4171113     // -----
4171114     // signed char,
4171115     // string index counter.
4171116     // -----
4171117     ptr_ptrdiff =
4171118         va_arg (ap, ptrdiff_t *);
4171119     *ptr_ptrdiff =
4171120         (ptrdiff_t) (input - start +
4171121                     scanned);
4171122     f += 2;
4171123 }
4171124 else
4171125 {
4171126     // -----
4171127     // unsupported or
4171128     // unknown specifier.
4171129     // -----
4171130     f += 1;
4171131 }
4171132 }
4171133 //

```

```

4171134 // Specifiers with no length modifier.
4171135 //
4171136 if (format[f] == 'd')
4171137 {
4171138     // ----- signed short, base 10.
4171139     value_i =
4171140         strtointmax (input, &next, 10, width);
4171141     if (input == next)
4171142     {
4171143         return (ass_or_eof
4171144             (consumed, assigned));
4171145     }
4171146     consumed++;
4171147     if (!flag_star)
4171148     {
4171149         ptr_sshort =
4171150             va_arg (ap, signed short *);
4171151         *ptr_sshort = value_i;
4171152         assigned++;
4171153     }
4171154     f += 1;
4171155     input = next;
4171156 }
4171157 else if (format[f] == 'i')
4171158 {
4171159     // -----
4171160     // signed
4171161     // int, base unknown.
4171162     // -----
4171163     value_i =
4171164         strtointmax (input, &next, 0, width);
4171165     if (input == next)
4171166     {
4171167         return (ass_or_eof
4171168             (consumed, assigned));
4171169     }
4171170     consumed++;
4171171     if (!flag_star)
4171172     {
4171173         ptr_sint = va_arg (ap, signed int *);
4171174         *ptr_sint = value_i;
4171175         assigned++;
4171176     }
4171177     f += 1;
4171178     input = next;
4171179 }
4171180 else if (format[f] == 'o')
4171181 {
4171182     // -----
4171183     // signed int, base 8.
4171184     // -----
4171185     value_i =
4171186         strtointmax (input, &next, 8, width);
4171187     if (input == next)
4171188     {
4171189         return (ass_or_eof
4171190             (consumed, assigned));
4171191     }
4171192     consumed++;
4171193     if (!flag_star)
4171194     {
4171195         ptr_sint = va_arg (ap, signed int *);
4171196         *ptr_sint = value_i;
4171197         assigned++;
4171198     }
4171199     f += 1;
4171200     input = next;
4171201 }
4171202 else if (format[f] == 'u')
4171203 {
4171204     // -----
4171205     // unsigned short, base 10.
4171206     // -----
4171207     value_u =
4171208         strtointmax (input, &next, 10, width);
4171209     if (input == next)
4171210     {
4171211         return (ass_or_eof
4171212             (consumed, assigned));
4171213     }
4171214     consumed++;
4171215     if (!flag_star)
4171216     {
4171217         ptr_uint =
4171218             va_arg (ap, unsigned int *);
4171219         *ptr_uint = value_u;
4171220         assigned++;

```

```

4171221     }
4171222     f += 1;
4171223     input = next;
4171224 }
4171225 else if (format[f] == 'x' || format[f] == 'X')
4171226 {
4171227     // -----
4171228     // signed short, base 16.
4171229     // -----
4171230     value_i =
4171231     strtointmax (input, &next, 16, width);
4171232     if (input == next)
4171233     {
4171234         return (ass_or_eof
4171235             (consumed, assigned));
4171236     }
4171237     consumed++;
4171238     if (!flag_star)
4171239     {
4171240         ptr_sint = va_arg (ap, signed int *);
4171241         *ptr_sint = value_i;
4171242         assigned++;
4171243     }
4171244     f += 1;
4171245     input = next;
4171246 }
4171247 else if (format[f] == 'c')
4171248 {
4171249     // ----- char[].
4171250     if (width == 0)
4171251         width = 1;
4171252     //
4171253     if (!flag_star)
4171254         ptr_char = va_arg (ap, char *);
4171255     //
4171256     for (count = 0;
4171257         width > 0 && *input != 0;
4171258         width--, ptr_char++, input++)
4171259     {
4171260         if (!flag_star)
4171261             *ptr_char = *input;
4171262         //
4171263         count++;
4171264     }
4171265     //
4171266     if (count)
4171267         consumed++;
4171268     if (count && !flag_star)
4171269         assigned++;
4171270     //
4171271     f += 1;
4171272 }
4171273 else if (format[f] == 's')
4171274 {
4171275     // ----- string.
4171276     if (!flag_star)
4171277         ptr_char = va_arg (ap, char *);
4171278     //
4171279     for (count = 0;
4171280         !isspace (*input)
4171281         && *input != 0; ptr_char++, input++)
4171282     {
4171283         if (!flag_star)
4171284             *ptr_char = *input;
4171285         //
4171286         count++;
4171287     }
4171288     if (!flag_star)
4171289         *ptr_char = 0;
4171290     //
4171291     if (count)
4171292         consumed++;
4171293     if (count && !flag_star)
4171294         assigned++;
4171295     //
4171296     f += 1;
4171297 }
4171298 else if (format[f] == '[')
4171299 {
4171300     //
4171301     f++;
4171302     //
4171303     if (format[f] == '^')
4171304     {
4171305         inverted = 1;
4171306         f++;
4171307     }

```

```

4171308     else
4171309     {
4171310         inverted = 0;
4171311     }
4171312     //
4171313     // Reset ascii array.
4171314     //
4171315     for (index = 0; index < 128; index++)
4171316     {
4171317         ascii[index] = inverted;
4171318     }
4171319     //
4171320     //
4171321     //
4171322     for (count = 0;
4171323         &format[f] < end_format; count++)
4171324     {
4171325         if (format[f] == ')' && count > 0)
4171326         {
4171327             break;
4171328         }
4171329         //
4171330         // Check for an interval.
4171331         //
4171332         if (format[f + 1] == '-'
4171333             && format[f + 2] != ')'
4171334             && format[f + 2] != 0)
4171335         {
4171336             //
4171337             // Interval.
4171338             //
4171339             for (index = format[f];
4171340                 index <= format[f + 2];
4171341                 index++)
4171342             {
4171343                 ascii[index] = !inverted;
4171344             }
4171345             f += 3;
4171346             continue;
4171347         }
4171348         //
4171349         // Single character.
4171350         //
4171351         index = format[f];
4171352         ascii[index] = !inverted;
4171353         f++;
4171354     }
4171355     //
4171356     // Is the scan correctly finished?.
4171357     //
4171358     if (format[f] != ']')
4171359     {
4171360         return (ass_or_eof
4171361             (consumed, assigned));
4171362     }
4171363     //
4171364     // The ascii table is populated.
4171365     //
4171366     if (width == 0)
4171367         width = SIZE_MAX;
4171368     //
4171369     // Scan the input string.
4171370     //
4171371     if (!flag_star)
4171372         ptr_char = va_arg (ap, char *);
4171373     //
4171374     for (count = 0;
4171375         width > 0 && *input != 0;
4171376         width--, ptr_char++, input++)
4171377     {
4171378         index = *input;
4171379         if (ascii[index])
4171380         {
4171381             if (!flag_star)
4171382                 *ptr_char = *input;
4171383             count++;
4171384         }
4171385         else
4171386         {
4171387             break;
4171388         }
4171389     }
4171390     //
4171391     if (count)
4171392         consumed++;
4171393     if (count && !flag_star)
4171394         assigned++;

```

```

4171395 //
4171396     f += 1;
4171397 }
4171398 else if (format[f] == 'p')
4171399 {
4171400     // ----- void *.
4171401     value_i =
4171402     strtointmax (input, &next, 16, width);
4171403     if (input == next)
4171404     {
4171405         return (ass_or_eof
4171406             (consumed, assigned));
4171407     }
4171408     consumed++;
4171409     if (!flag_star)
4171410     {
4171411         ptr_void = va_arg (ap, void **);
4171412         *ptr_void = (void *) ((int) value_i);
4171413         assigned++;
4171414     }
4171415     f += 1;
4171416     input = next;
4171417 }
4171418 else if (format[f] == 'n')
4171419 {
4171420     // -----
4171421     // signed char,
4171422     // string index counter.
4171423     // -----
4171424     ptr_sint = va_arg (ap, signed int *);
4171425     *ptr_sint =
4171426     (signed char) (input - start + scanned);
4171427     f += 1;
4171428 }
4171429 else
4171430 {
4171431     // -----
4171432     // unsupported or
4171433     // unknown specifier.
4171434     // -----
4171435     ;
4171436 }
4171437
4171438 // -----
4171439 // End of specifier.
4171440 // -----
4171441
4171442 width_string[0] = '\0';
4171443 specifier = 0;
4171444 specifier_flags = 0;
4171445 specifier_width = 0;
4171446 specifier_type = 0;
4171447 flag_star = 0;
4171448 }
4171449 }
4171450 }
4171451 //
4171452 // The format or the input string is terminated.
4171453 //
4171454 if (&format[f] < end_format && stream)
4171455 {
4171456     //
4171457     // Only the input string is finished, and
4171458     // the input comes
4171459     // from a stream, so another read will be
4171460     // done.
4171461     //
4171462     scanned += (int) (input - start);
4171463     continue;
4171464 }
4171465 //
4171466 // The format string is terminated.
4171467 //
4171468 return (ass_or_eof (consumed, assigned));
4171469 }
4171470 }
4171471
4171472 //-----
4171473 static intmax_t
4171474 strtointmax (const char *restrict string,
4171475             const char **restrict endptr, int base,
4171476             size_t max_width)
4171477 {
4171478     int i;
4171479     int d; // Digits counter.
4171480     int sign = +1;
4171481     intmax_t number;

```

```

4171482     intmax_t previous;
4171483     int digit;
4171484     //
4171485     bool flag_prefix_oct = 0;
4171486     bool flag_prefix_exa = 0;
4171487     bool flag_prefix_dec = 0;
4171488     //
4171489     // If the 'max_width' value is zero, fix it to the
4171490     // maximum
4171491     // that it can represent.
4171492     //
4171493     if (max_width == 0)
4171494     {
4171495         max_width = SIZE_MAX;
4171496     }
4171497     //
4171498     // Eat initial spaces, but if there are spaces,
4171499     // there is an
4171500     // error inside the calling function!
4171501     //
4171502     for (i = 0; isspace (string[i]); i++)
4171503     {
4171504         fprintf (stderr,
4171505             "libc error: file \"%s\", line %i\n",
4171506             __FILE__, __LINE__);
4171507     }
4171508     //
4171509     //
4171510     // Check sign. The 'max_width' counts also the sign,
4171511     // if there is
4171512     // one.
4171513     //
4171514     if (string[i] == '+')
4171515     {
4171516         sign = +1;
4171517         i++;
4171518         max_width--;
4171519     }
4171520     else if (string[i] == '-')
4171521     {
4171522         sign = -1;
4171523         i++;
4171524         max_width--;
4171525     }
4171526     //
4171527     // Check for prefix.
4171528     //
4171529     if (string[i] == '0')
4171530     {
4171531         if (string[i + 1] == 'x' || string[i + 1] == 'X')
4171532         {
4171533             flag_prefix_exa = 1;
4171534         }
4171535         if (isdigit (string[i + 1]))
4171536         {
4171537             flag_prefix_oct = 1;
4171538         }
4171539     }
4171540     //
4171541     if (string[i] > '0' && string[i] <= '9')
4171542     {
4171543         flag_prefix_dec = 1;
4171544     }
4171545     //
4171546     // Check compatibility with requested base.
4171547     //
4171548     if (flag_prefix_exa)
4171549     {
4171550         if (base == 0)
4171551         {
4171552             base = 16;
4171553         }
4171554         else if (base == 16)
4171555         {
4171556             ; // Ok.
4171557         }
4171558         else
4171559         {
4171560             //
4171561             // Incompatible sequence: only the initial
4171562             // zero is reported.
4171563             //
4171564             *endptr = &string[i + 1];
4171565             return ((intmax_t) 0);
4171566         }
4171567     }
4171568     //
4171569     // Move on, after the '0x' prefix.

```

```

4171569 //
4171570 i += 2;
4171571 }
4171572 //
4171573 if (flag_prefix_oct)
4171574 {
4171575     if (base == 0)
4171576     {
4171577         base = 8;
4171578     }
4171579 //
4171580 // Move on, after the '0' prefix.
4171581 //
4171582 i += 1;
4171583 }
4171584 //
4171585 if (flag_prefix_dec)
4171586 {
4171587     if (base == 0)
4171588     {
4171589         base = 10;
4171590     }
4171591 }
4171592 //
4171593 // Scan the string.
4171594 //
4171595 for (d = 0, number = 0;
4171596      d < max_width && string[i] != 0; i++, d++)
4171597 {
4171598     if (string[i] >= '0' && string[i] <= '9')
4171599     {
4171600         digit = string[i] - '0';
4171601     }
4171602     else if (string[i] >= 'A' && string[i] <= 'F')
4171603     {
4171604         digit = string[i] - 'A' + 10;
4171605     }
4171606     else if (string[i] >= 'a' && string[i] <= 'f')
4171607     {
4171608         digit = string[i] - 'a' + 10;
4171609     }
4171610     else
4171611     {
4171612         digit = 999;
4171613     }
4171614 //
4171615 // Give a sign to the digit.
4171616 //
4171617 digit *= sign;
4171618 //
4171619 // Compare with the base.
4171620 //
4171621 if (base > (digit * sign))
4171622 {
4171623     //
4171624     // Check if the current digit can be safely
4171625     // computed.
4171626     //
4171627     previous = number;
4171628     number *= base;
4171629     number += digit;
4171630     if (number / base != previous)
4171631     {
4171632         //
4171633         // Out of range.
4171634         //
4171635         *endptr = &string[i + 1];
4171636         errset (ERANGE); // Result too large.
4171637         if (sign > 0)
4171638         {
4171639             return (INTMAX_MAX);
4171640         }
4171641         else
4171642         {
4171643             return (INTMAX_MIN);
4171644         }
4171645     }
4171646 }
4171647 else
4171648 {
4171649     *endptr = &string[i];
4171650     return (number);
4171651 }
4171652 }
4171653 //
4171654 // The string is finished or the max digits length
4171655 // is reached.

```

```

4171656 //
4171657 *endptr = &string[i];
4171658 //
4171659 return (number);
4171660 }
4171661 //-----
4171662 static int
4171663 ass_or_eof (int consumed, int assigned)
4171664 {
4171665     if (consumed == 0)
4171666     {
4171667         return (EOF);
4171668     }
4171669     else
4171670     {
4171671         return (assigned);
4171672     }
4171673 }
4171674 //-----
4171675

```

## 95.18.40 lib/stdio/vprintf.c

Si veda la sezione 88.137.

```

4180001 #include <stdio.h>
4180002 #include <sys/types.h>
4180003 #include <sys/os32.h>
4180004 #include <string.h>
4180005 #include <unistd.h>
4180006 //-----
4180007 int
4180008 vprintf (const char *restrict format, va_list arg)
4180009 {
4180010     ssize_t size_written;
4180011     size_t size;
4180012     size_t size_total;
4180013     int status;
4180014     char string[BUFSIZ];
4180015     char *buffer = string;
4180016
4180017     buffer[0] = 0;
4180018     status = vsprintf (buffer, format, arg);
4180019
4180020     size = strlen (buffer);
4180021     if (size >= BUFSIZ)
4180022     {
4180023         size = BUFSIZ;
4180024     }
4180025
4180026     for (size_total = 0, size_written = 0;
4180027         size_total < size;
4180028         size_total += size_written, buffer += size_written)
4180029     {
4180030         //
4180031         // Write to the standard output: file descriptor
4180032         // n. 1.
4180033         //
4180034         size_written =
4180035             write (STDOUT_FILENO, buffer, size - size_total);
4180036         if (size_written < 0)
4180037         {
4180038             return (size_total);
4180039         }
4180040     }
4180041     return (size);
4180042 }

```

## 95.18.41 lib/stdio/vscanf.c

Si veda la sezione 88.138.

```

4190001 #include <stdio.h>
4190002 //-----
4190003 int
4190004 vscanf (const char *restrict format, va_list ap)
4190005 {
4190006     return (vfscanf (stdin, format, ap));
4190007 }
4190008 //-----
4190009

```



95.18.42 lib/stdio/vsnprintf.c

Si veda la sezione 88.137.

```

420001 #include <stdint.h>
420002 #include <stdbool.h>
420003 #include <stdlib.h>
420004 #include <string.h>
420005 #include <stdio.h>
420006 //-----
420007 static size_t uimaxtoa (uintmax_t integer,
420008                       char *buffer, int base,
420009                       int uppercase, size_t size);
420010 static size_t imaxtoa (intmax_t integer, char *buffer,
420011                      int base, int uppercase,
420012                      size_t size);
420013 static size_t simaxtoa (intmax_t integer, char *buffer,
420014                       int base, int uppercase,
420015                       size_t size);
420016 static size_t uimaxtoa_fill (uintmax_t integer,
420017                             char *buffer, int base,
420018                             int uppercase, int width,
420019                             int filler, int max);
420020 static size_t imaxtoa_fill (intmax_t integer,
420021                             char *buffer, int base,
420022                             int uppercase, int width,
420023                             int filler, int max);
420024 static size_t simaxtoa_fill (intmax_t integer,
420025                             char *buffer, int base,
420026                             int uppercase, int width,
420027                             int filler, int max);
420028 static size_t strtostri_fill (char *string,
420029                              char *buffer, int width,
420030                              int filler, int max);
420031 //-----
420032 int
420033 vsnprintf (char *restrict string, size_t size,
420034           const char *restrict format, va_list ap)
420035 {
420036     //
420037     // We produce at most 'size-1' characters, + '\0'.
420038     // 'size' is used also as the max size for internal
420039     // strings, but only if it is not too big.
420040     //
420041     int f = 0;
420042     int s = 0;
420043     int remain = size - 1;
420044     //
420045     bool specifier = 0;
420046     bool specifier_flags = 0;
420047     bool specifier_width = 0;
420048     bool specifier_precision = 0;
420049     bool specifier_type = 0;
420050     //
420051     bool flag_plus = 0;
420052     bool flag_minus = 0;
420053     bool flag_space = 0;
420054     bool flag_alternate = 0;
420055     bool flag_zero = 0;
420056     //
420057     int alignment;
420058     int filler;
420059     //
420060     intmax_t value_i;
420061     uintmax_t value_ui;
420062     char *value_cp;
420063     //
420064     size_t width;
420065     size_t precision;
420066     size_t str_size =
420067         (size > (BUFSIZ / 2) ? (BUFSIZ / 2) : size);
420068     char width_string[str_size];
420069     char precision_string[str_size];
420070     int w;
420071     int p;
420072     //
420073     width_string[0] = '\0';
420074     precision_string[0] = '\0';
420075     //
420076     while (format[f] != 0 && s < (size - 1))
420077     {
420078         if (!specifier)
420079         {
420080             // ----- The context is not
420081             // inside a specifier.
420082             if (format[f] != '%')
420083             {
420084                 string[s] = format[f];
420085                 s++;

```

```

420086         remain--;
420087         f++;
420088         continue;
420089     }
420090     if (format[f] == '%' && format[f + 1] == '%')
420091     {
420092         string[s] = '%';
420093         f++;
420094         f++;
420095         s++;
420096         remain--;
420097         continue;
420098     }
420099     if (format[f] == '%')
420100     {
420101         f++;
420102         specifier = 1;
420103         specifier_flags = 1;
420104         continue;
420105     }
420106 }
420107 //
420108 if (specifier && specifier_flags)
420109 {
420110     // ----- The context is inside
420111     // specifier flags.
420112     if (format[f] == '+')
420113     {
420114         flag_plus = 1;
420115         f++;
420116         continue;
420117     }
420118     else if (format[f] == '-')
420119     {
420120         flag_minus = 1;
420121         f++;
420122         continue;
420123     }
420124     else if (format[f] == ' ')
420125     {
420126         flag_space = 1;
420127         f++;
420128         continue;
420129     }
420130     else if (format[f] == '#')
420131     {
420132         flag_alternate = 1;
420133         f++;
420134         continue;
420135     }
420136     else if (format[f] == '0')
420137     {
420138         flag_zero = 1;
420139         f++;
420140         continue;
420141     }
420142     else
420143     {
420144         specifier_flags = 0;
420145         specifier_width = 1;
420146     }
420147 }
420148 //
420149 if (specifier && specifier_width)
420150 {
420151     // ----- The context is inside
420152     // specifier width.
420153     for (w = 0;
420154          format[f] >= '0' && format[f] <= '9'
420155          && w < str_size; w++)
420156     {
420157         width_string[w] = format[f];
420158         f++;
420159     }
420160     width_string[w] = '\0';
420161
420162     specifier_width = 0;
420163
420164     if (format[f] == '.')
420165     {
420166         specifier_precision = 1;
420167         f++;
420168     }
420169     else
420170     {
420171         specifier_precision = 0;
420172         specifier_type = 1;

```

```

4200173     }
4200174     }
4200175     //
4200176     if (specifier && specifier_precision)
4200177     {
4200178         // ----- The context is inside
4200179         // specifier precision.
4200180         for (p = 0;
4200181             format[f] >= '0' && format[f] <= '9'
4200182             && p < str_size; p++)
4200183         {
4200184             precision_string[p] = format[f];
4200185             p++;
4200186         }
4200187         precision_string[p] = '\0';
4200188
4200189         specifier_precision = 0;
4200190         specifier_type = 1;
4200191     }
4200192     //
4200193     if (specifier && specifier_type)
4200194     {
4200195         // ----- The context is
4200196         // inside specifier type.
4200197         width = atoi (width_string);
4200198         precision = atoi (precision_string);
4200199         filler = ' ';
4200200         if (flag_zero)
4200201             filler = '0';
4200202         if (flag_space)
4200203             filler = ' ';
4200204         alignment = width;
4200205         if (flag_minus)
4200206         {
4200207             alignment = -alignment;
4200208             filler = ' '; // The filler
4200209             // character cannot
4200210             // be zero, so it is black.
4200211         }
4200212         //
4200213         if (format[f] == 'h' && format[f + 1] == 'h')
4200214         {
4200215             if (format[f + 2] == 'd'
4200216                 || format[f + 2] == 'i')
4200217             {
4200218                 // -----
4200219                 // signed char, base 10.
4200220                 value_i = va_arg (ap, int);
4200221                 if (flag_plus)
4200222                 {
4200223                     s +=
4200224                         simaxtoa_fill (value_i,
4200225                                         &string[s], 10,
4200226                                         0, alignment,
4200227                                         filler, remain);
4200228                 }
4200229                 else
4200230                 {
4200231                     s +=
4200232                         imaxtoa_fill (value_i,
4200233                                       &string[s], 10,
4200234                                       0, alignment,
4200235                                       filler, remain);
4200236                 }
4200237                 f += 3;
4200238             }
4200239             else if (format[f + 2] == 'u')
4200240             {
4200241                 // -----
4200242                 // unsigned char, base 10.
4200243                 value_ui = va_arg (ap, unsigned int);
4200244                 s +=
4200245                     uimaxtoa_fill (value_ui,
4200246                                    &string[s], 10, 0,
4200247                                    alignment, filler,
4200248                                    remain);
4200249                 f += 3;
4200250             }
4200251             else if (format[f + 2] == 'o')
4200252             {
4200253                 // -----
4200254                 // unsigned char, base 8.
4200255                 value_ui = va_arg (ap, unsigned int);
4200256                 s +=
4200257                     uimaxtoa_fill (value_ui,
4200258                                    &string[s], 8, 0,
4200259                                    alignment, filler,

```

```

4200260             remain);
4200261             f += 3;
4200262         }
4200263         else if (format[f + 2] == 'x')
4200264         {
4200265             // -----
4200266             // unsigned char, base 16.
4200267             value_ui = va_arg (ap, unsigned int);
4200268             s +=
4200269                 uimaxtoa_fill (value_ui,
4200270                                &string[s], 16, 0,
4200271                                alignment, filler,
4200272                                remain);
4200273             f += 3;
4200274         }
4200275         else if (format[f + 2] == 'X')
4200276         {
4200277             // -----
4200278             // unsigned char, base 16.
4200279             value_ui = va_arg (ap, unsigned int);
4200280             s +=
4200281                 uimaxtoa_fill (value_ui,
4200282                                &string[s], 16, 1,
4200283                                alignment, filler,
4200284                                remain);
4200285             f += 3;
4200286         }
4200287         else if (format[f + 2] == 'b')
4200288         {
4200289             // ----- unsigned char,
4200290             // base 2 (extention).
4200291             value_ui = va_arg (ap, unsigned int);
4200292             s +=
4200293                 uimaxtoa_fill (value_ui,
4200294                                &string[s], 2, 0,
4200295                                alignment, filler,
4200296                                remain);
4200297             f += 3;
4200298         }
4200299         else
4200300         {
4200301             // ----- unsupported or
4200302             // unknown specifier.
4200303             f += 2;
4200304         }
4200305     }
4200306     else if (format[f] == 'h')
4200307     {
4200308         if (format[f + 1] == 'd'
4200309             || format[f + 1] == 'i')
4200310         {
4200311             // -----
4200312             // short int, base 10.
4200313             value_i = va_arg (ap, int);
4200314             if (flag_plus)
4200315             {
4200316                 s +=
4200317                     simaxtoa_fill (value_i,
4200318                                    &string[s], 10,
4200319                                    0, alignment,
4200320                                    filler, remain);
4200321             }
4200322             else
4200323             {
4200324                 s +=
4200325                     imaxtoa_fill (value_i,
4200326                                    &string[s], 10,
4200327                                    0, alignment,
4200328                                    filler, remain);
4200329             }
4200330             f += 2;
4200331         }
4200332         else if (format[f + 1] == 'u')
4200333         {
4200334             // ----- unsigned
4200335             // short int, base 10.
4200336             value_ui = va_arg (ap, unsigned int);
4200337             s +=
4200338                 uimaxtoa_fill (value_ui,
4200339                                &string[s], 10, 0,
4200340                                alignment, filler,
4200341                                remain);
4200342             f += 2;
4200343         }
4200344         else if (format[f + 1] == 'o')
4200345         {
4200346             // ----- unsigned

```

```

4200347 // short int, base 8.
4200348 value_ui = va_arg (ap, unsigned int);
4200349 s +=
4200350     uimaxtoa_fill (value_ui,
4200351                   &string[s], 8, 0,
4200352                   alignment, filler,
4200353                   remain);
4200354     f += 2;
4200355 }
4200356 else if (format[f + 1] == 'x')
4200357 {
4200358 // ----- unsigned
4200359 // short int, base 16.
4200360 value_ui = va_arg (ap, unsigned int);
4200361 s +=
4200362     uimaxtoa_fill (value_ui,
4200363                   &string[s], 16, 0,
4200364                   alignment, filler,
4200365                   remain);
4200366     f += 2;
4200367 }
4200368 else if (format[f + 1] == 'X')
4200369 {
4200370 // ----- unsigned
4200371 // short int, base 16.
4200372 value_ui = va_arg (ap, unsigned int);
4200373 s +=
4200374     uimaxtoa_fill (value_ui,
4200375                   &string[s], 16, 1,
4200376                   alignment, filler,
4200377                   remain);
4200378     f += 2;
4200379 }
4200380 else if (format[f + 1] == 'b')
4200381 {
4200382 // ----- unsigned short int,
4200383 // base 2 (extention).
4200384 value_ui = va_arg (ap, unsigned int);
4200385 s +=
4200386     uimaxtoa_fill (value_ui,
4200387                   &string[s], 2, 0,
4200388                   alignment, filler,
4200389                   remain);
4200390     f += 2;
4200391 }
4200392 else
4200393 {
4200394 // ----- unsupported or
4200395 // unknown specifier.
4200396     f += 1;
4200397 }
4200398 }
4200399 else if (format[f] == 'l' && format[f + 1] != 'l')
4200400 {
4200401     if (format[f + 1] == 'd'
4200402         || format[f + 1] == 'i')
4200403     {
4200404 // -----
4200405 // long int base 10.
4200406 value_i = va_arg (ap, long int);
4200407 if (flag_plus)
4200408     {
4200409         s +=
4200410             simaxtoa_fill (value_i,
4200411                           &string[s], 10,
4200412                           0, alignment,
4200413                           filler, remain);
4200414     }
4200415     else
4200416     {
4200417         s +=
4200418             imaxtoa_fill (value_i,
4200419                           &string[s], 10,
4200420                           0, alignment,
4200421                           filler, remain);
4200422     }
4200423     f += 2;
4200424 }
4200425 else if (format[f + 1] == 'u')
4200426 {
4200427 // ----- Unsigned
4200428 // long int base 10.
4200429 value_ui = va_arg (ap, unsigned long int);
4200430 s +=
4200431     uimaxtoa_fill (value_ui,
4200432                   &string[s], 10, 0,
4200433                   alignment, filler,

```

```

4200434         remain);
4200435     f += 2;
4200436 }
4200437 else if (format[f + 1] == 'o')
4200438 {
4200439 // ----- Unsigned
4200440 // long int base 8.
4200441 value_ui = va_arg (ap, unsigned long int);
4200442 s +=
4200443     uimaxtoa_fill (value_ui,
4200444                   &string[s], 8, 0,
4200445                   alignment, filler,
4200446                   remain);
4200447     f += 2;
4200448 }
4200449 else if (format[f + 1] == 'x')
4200450 {
4200451 // ----- Unsigned
4200452 // long int base 16.
4200453 value_ui = va_arg (ap, unsigned long int);
4200454 s +=
4200455     uimaxtoa_fill (value_ui,
4200456                   &string[s], 16, 0,
4200457                   alignment, filler,
4200458                   remain);
4200459     f += 2;
4200460 }
4200461 else if (format[f + 1] == 'X')
4200462 {
4200463 // ----- Unsigned
4200464 // long int base 16.
4200465 value_ui = va_arg (ap, unsigned long int);
4200466 s +=
4200467     uimaxtoa_fill (value_ui,
4200468                   &string[s], 16, 1,
4200469                   alignment, filler,
4200470                   remain);
4200471     f += 2;
4200472 }
4200473 else if (format[f + 1] == 'b')
4200474 {
4200475 // ----- Unsigned long int
4200476 // base 2 (extention).
4200477 value_ui = va_arg (ap, unsigned long int);
4200478 s +=
4200479     uimaxtoa_fill (value_ui,
4200480                   &string[s], 2, 0,
4200481                   alignment, filler,
4200482                   remain);
4200483     f += 2;
4200484 }
4200485 else
4200486 {
4200487 // ----- unsupported or
4200488 // unknown specifier.
4200489     f += 1;
4200490 }
4200491 }
4200492 else if (format[f] == 'l' && format[f + 1] == 'l')
4200493 {
4200494     if (format[f + 2] == 'd'
4200495         || format[f + 2] == 'i')
4200496     {
4200497 // -----
4200498 // long int base 10.
4200499 value_i = va_arg (ap, long long int);
4200500 if (flag_plus)
4200501     {
4200502         s +=
4200503             simaxtoa_fill (value_i,
4200504                           &string[s], 10,
4200505                           0, alignment,
4200506                           filler, remain);
4200507     }
4200508     else
4200509     {
4200510         s +=
4200511             imaxtoa_fill (value_i,
4200512                           &string[s], 10,
4200513                           0, alignment,
4200514                           filler, remain);
4200515     }
4200516     f += 3;
4200517 }
4200518 else if (format[f + 2] == 'u')
4200519 {
4200520 // ----- Unsigned

```

```

420021 // long int base 10.
420022 value_ui =
420023 va_arg (ap, unsigned long long int);
420024 s +=
420025 uimaxtoa_fill (value_ui,
420026                &string[s], 10, 0,
420027                alignment, filler,
420028                remain);
420029
420030 f += 3;
420031 }
420032 else if (format[f + 2] == 'o')
420033 {
420034 // ----- Unsigned
420035 // long int base 8.
420036 value_ui =
420037 va_arg (ap, unsigned long long int);
420038 s +=
420039 uimaxtoa_fill (value_ui,
420040                &string[s], 8, 0,
420041                alignment, filler,
420042                remain);
420043
420044 f += 3;
420045 }
420046 else if (format[f + 2] == 'x')
420047 {
420048 // ----- Unsigned
420049 // long int base 16.
420050 value_ui =
420051 va_arg (ap, unsigned long long int);
420052 s +=
420053 uimaxtoa_fill (value_ui,
420054                &string[s], 16, 0,
420055                alignment, filler,
420056                remain);
420057
420058 f += 3;
420059 }
420060 else if (format[f + 2] == 'X')
420061 {
420062 // ----- Unsigned
420063 // long int base 16.
420064 value_ui =
420065 va_arg (ap, unsigned long long int);
420066 s +=
420067 uimaxtoa_fill (value_ui,
420068                &string[s], 16, 1,
420069                alignment, filler,
420070                remain);
420071
420072 f += 3;
420073 }
420074 else if (format[f + 2] == 'b')
420075 {
420076 // ----- Unsigned long int
420077 // base 2 (extention).
420078 value_ui =
420079 va_arg (ap, unsigned long long int);
420080 s +=
420081 uimaxtoa_fill (value_ui,
420082                &string[s], 2, 0,
420083                alignment, filler,
420084                remain);
420085
420086 f += 3;
420087 }
420088 else
420089 {
420090 // ----- unsupported or
420091 // unknown specifier.
420092 f += 2;
420093 }
420094 else if (format[f] == 'j')
420095 {
420096 if (format[f + 1] == 'd'
420097     || format[f + 1] == 'i')
420098 {
420099 // -----
420100 // intmax_t base 10.
420101 value_i = va_arg (ap, intmax_t);
420102 if (flag_plus)
420103 {
420104 s +=
420105 simaxtoa_fill (value_i,
420106                &string[s], 10,
420107                0, alignment,
420108                filler, remain);
420109 }
420110 else
420111 {

```

```

420082 s +=
420083 imaxtoa_fill (value_i,
420084               &string[s], 10,
420085               0, alignment,
420086               filler, remain);
420087 }
420088 f += 2;
420089 }
420090 else if (format[f + 1] == 'u')
420091 {
420092 // -----
420093 // uintmax_t base 10.
420094 value_ui = va_arg (ap, uintmax_t);
420095 s +=
420096 uimaxtoa_fill (value_ui,
420097                &string[s], 10, 0,
420098                alignment, filler,
420099                remain);
420100
420101 f += 2;
420102 }
420103 else if (format[f + 1] == 'o')
420104 {
420105 // -----
420106 // uintmax_t base 8.
420107 value_ui = va_arg (ap, uintmax_t);
420108 s +=
420109 uimaxtoa_fill (value_ui,
420110                &string[s], 8, 0,
420111                alignment, filler,
420112                remain);
420113
420114 f += 2;
420115 }
420116 else if (format[f + 1] == 'x')
420117 {
420118 // -----
420119 // uintmax_t base 16.
420120 value_ui = va_arg (ap, uintmax_t);
420121 s +=
420122 uimaxtoa_fill (value_ui,
420123                &string[s], 16, 0,
420124                alignment, filler,
420125                remain);
420126
420127 f += 2;
420128 }
420129 else if (format[f + 1] == 'X')
420130 {
420131 // -----
420132 // uintmax_t base 16.
420133 value_ui = va_arg (ap, uintmax_t);
420134 s +=
420135 uimaxtoa_fill (value_ui,
420136                &string[s], 16, 1,
420137                alignment, filler,
420138                remain);
420139
420140 f += 2;
420141 }
420142 else if (format[f + 1] == 'b')
420143 {
420144 // ----- uintmax_t
420145 // base 2 (extention).
420146 value_ui = va_arg (ap, uintmax_t);
420147 s +=
420148 uimaxtoa_fill (value_ui,
420149                &string[s], 2, 0,
420150                alignment, filler,
420151                remain);
420152
420153 f += 2;
420154 }
420155 else
420156 {
420157 // ----- unsupported or
420158 // unknown specifier.
420159 f += 1;
420160 }
420161 }
420162 else if (format[f] == 'z')
420163 {
420164 if (format[f + 1] == 'd'
420165     || format[f + 1] == 'i'
420166     || format[f + 1] == 'i')
420167 {
420168 // ----- size_t base 10.
420169 value_ui = va_arg (ap, unsigned long int);
420170 s +=
420171 uimaxtoa_fill (value_ui,
420172                &string[s], 10, 0,
420173                alignment, filler,

```

```

4200695         remain);
4200696     f += 2;
4200697 }
4200698 else if (format[f + 1] == 'o')
4200699 {
4200700     // ----- size_t base 8.
4200701     value_ui = va_arg (ap, unsigned long int);
4200702     s +=
4200703         uimaxtoa_fill (value_ui,
4200704                       &string[s], 8, 0,
4200705                       alignment, filler,
4200706                       remain);
4200707     f += 2;
4200708 }
4200709 else if (format[f + 1] == 'x')
4200710 {
4200711     // ----- size_t base 16.
4200712     value_ui = va_arg (ap, unsigned long int);
4200713     s +=
4200714         uimaxtoa_fill (value_ui,
4200715                       &string[s], 16, 0,
4200716                       alignment, filler,
4200717                       remain);
4200718     f += 2;
4200719 }
4200720 else if (format[f + 1] == 'X')
4200721 {
4200722     // ----- size_t base 16.
4200723     value_ui = va_arg (ap, unsigned long int);
4200724     s +=
4200725         uimaxtoa_fill (value_ui,
4200726                       &string[s], 16, 1,
4200727                       alignment, filler,
4200728                       remain);
4200729     f += 2;
4200730 }
4200731 else if (format[f + 1] == 'b')
4200732 {
4200733     // ----- size_t
4200734     // base 2 (extension).
4200735     value_ui = va_arg (ap, unsigned long int);
4200736     s +=
4200737         uimaxtoa_fill (value_ui,
4200738                       &string[s], 2, 0,
4200739                       alignment, filler,
4200740                       remain);
4200741     f += 2;
4200742 }
4200743 else
4200744 {
4200745     // ----- unsupported or
4200746     // unknown specifier.
4200747     f += 1;
4200748 }
4200749 }
4200750 else if (format[f] == 't')
4200751 {
4200752     if (format[f + 1] == 'd'
4200753         || format[f + 1] == 'i')
4200754     {
4200755         // -----
4200756         // ptrdiff_t base 10.
4200757         value_i = va_arg (ap, long int);
4200758         if (flag_plus)
4200759         {
4200760             s +=
4200761                 simaxtoa_fill (value_i,
4200762                               &string[s], 10,
4200763                               0, alignment,
4200764                               filler, remain);
4200765         }
4200766         else
4200767         {
4200768             s +=
4200769                 imaxtoa_fill (value_i,
4200770                              &string[s], 10,
4200771                              0, alignment,
4200772                              filler, remain);
4200773         }
4200774         f += 2;
4200775     }
4200776     else if (format[f + 1] == 'u')
4200777     {
4200778         // ----- ptrdiff_t base
4200779         // 10, without sign.
4200780         value_ui = va_arg (ap, unsigned long int);
4200781         s +=

```

```

4200782         uimaxtoa_fill (value_ui,
4200783                       &string[s], 10, 0,
4200784                       alignment, filler,
4200785                       remain);
4200786     f += 2;
4200787 }
4200788 else if (format[f + 1] == 'o')
4200789 {
4200790     // ----- ptrdiff_t base
4200791     // 8, without sign.
4200792     value_ui = va_arg (ap, unsigned long int);
4200793     s +=
4200794         uimaxtoa_fill (value_ui,
4200795                       &string[s], 8, 0,
4200796                       alignment, filler,
4200797                       remain);
4200798     f += 2;
4200799 }
4200800 else if (format[f + 1] == 'x')
4200801 {
4200802     // ----- ptrdiff_t base
4200803     // 16, without sign.
4200804     value_ui = va_arg (ap, unsigned long int);
4200805     s +=
4200806         uimaxtoa_fill (value_ui,
4200807                       &string[s], 16, 0,
4200808                       alignment, filler,
4200809                       remain);
4200810     f += 2;
4200811 }
4200812 else if (format[f + 1] == 'X')
4200813 {
4200814     // ----- ptrdiff_t base
4200815     // 16, without sign.
4200816     value_ui = va_arg (ap, unsigned long int);
4200817     s +=
4200818         uimaxtoa_fill (value_ui,
4200819                       &string[s], 16, 1,
4200820                       alignment, filler,
4200821                       remain);
4200822     f += 2;
4200823 }
4200824 else if (format[f + 1] == 'b')
4200825 {
4200826     // ----- ptrdiff_t base 2, without
4200827     // sign (extension).
4200828     value_ui = va_arg (ap, unsigned long int);
4200829     s +=
4200830         uimaxtoa_fill (value_ui,
4200831                       &string[s], 2, 0,
4200832                       alignment, filler,
4200833                       remain);
4200834     f += 2;
4200835 }
4200836 else
4200837 {
4200838     // ----- unsupported or
4200839     // unknown specifier.
4200840     f += 1;
4200841 }
4200842 }
4200843 if (format[f] == 'd' || format[f] == 'i')
4200844 {
4200845     // ----- int base 10.
4200846     value_i = va_arg (ap, int);
4200847     if (flag_plus)
4200848     {
4200849         s +=
4200850             simaxtoa_fill (value_i, &string[s],
4200851                           10, 0, alignment,
4200852                           filler, remain);
4200853     }
4200854     else
4200855     {
4200856         s +=
4200857             imaxtoa_fill (value_i, &string[s],
4200858                           10, 0, alignment,
4200859                           filler, remain);
4200860     }
4200861     f += 1;
4200862 }
4200863 else if (format[f] == 'u')
4200864 {
4200865     // -----
4200866     // unsigned int base 10.
4200867     value_ui = va_arg (ap, unsigned int);
4200868     s +=

```

```

420069     uimaxtoa_fill (value_ui, &string[s],
420070                  10, 0, alignment,
420071                  filler, remain);
420072
420073     f += 1;
420074 }
420075 else if (format[f] == 'o')
420076 {
420077     // ----- unsigned int base 8.
420078     value_ui = va_arg (ap, unsigned int);
420079     s +=
420080         uimaxtoa_fill (value_ui, &string[s], 8,
420081                       0, alignment, filler,
420082                       remain);
420083     f += 1;
420084 }
420085 else if (format[f] == 'x')
420086 {
420087     // -----
420088     // unsigned int base 16.
420089     value_ui = va_arg (ap, unsigned int);
420090     s +=
420091         uimaxtoa_fill (value_ui, &string[s],
420092                       16, 0, alignment,
420093                       filler, remain);
420094     f += 1;
420095 }
420096 else if (format[f] == 'X')
420097 {
420098     // -----
420099     // unsigned int base 16.
420100     value_ui = va_arg (ap, unsigned int);
420101     s +=
420102         uimaxtoa_fill (value_ui, &string[s],
420103                       16, 1, alignment,
420104                       filler, remain);
420105     f += 1;
420106 }
420107 else if (format[f] == 'b')
420108 {
420109     // ----- unsigned int
420110     // base 2 (extention).
420111     value_ui = va_arg (ap, unsigned int);
420112     s +=
420113         uimaxtoa_fill (value_ui, &string[s], 2,
420114                       0, alignment, filler,
420115                       remain);
420116     f += 1;
420117 }
420118 else if (format[f] == 'c')
420119 {
420120     // ----- unsigned char.
420121     value_ui = va_arg (ap, unsigned int);
420122     string[s] = (char) value_ui;
420123     s += 1;
420124     f += 1;
420125 }
420126 else if (format[f] == 's')
420127 {
420128     // ----- string.
420129     value_cp = va_arg (ap, char *);
420130     filler = ' ';
420131
420132     s +=
420133         strtostr_fill (value_cp, &string[s],
420134                      alignment, filler, remain);
420135     f += 1;
420136 }
420137 else
420138 {
420139     // ----- unsupported or
420140     // unknown specifier.
420141     ;
420142 }
420143 // -----
420144 // End of specifier.
420145 // -----
420146 width_string[0] = '\0';
420147 precision_string[0] = '\0';
420148
420149 specifier = 0;
420150 specifier_flags = 0;
420151 specifier_width = 0;
420152 specifier_precision = 0;
420153 specifier_type = 0;
420154
420155 flag_plus = 0;
420156 flag_minus = 0;

```

```

420056     flag_space = 0;
420057     flag_alternate = 0;
420058     flag_zero = 0;
420059 }
420060 }
420061 string[s] = '\0';
420062 return s;
420063 }
420064
420065 //-----
420066 // Static functions.
420067 //-----
420068 static size_t
420069 uimaxtoa (uintmax_t integer, char *buffer, int base,
420070           int uppercase, size_t size)
420071 {
420072     // -----
420073     // Convert a maximum rank integer into a string.
420074     // -----
420075
420076     uintmax_t integer_copy = integer;
420077     size_t digits;
420078     int b;
420079     unsigned char remainder;
420080
420081     for (digits = 0; integer_copy > 0; digits++)
420082     {
420083         integer_copy = integer_copy / base;
420084     }
420085
420086     if (buffer == NULL && integer == 0)
420087         return 1;
420088     if (buffer == NULL && integer > 0)
420089         return digits;
420090
420091     if (integer == 0)
420092     {
420093         buffer[0] = '0';
420094         buffer[1] = '\0';
420095         return 1;
420096     }
420097     //
420098     // Fix the maximum number of digits.
420099     //
420100     if (size > 0 && digits > size)
420101         digits = size;
420102     //
420103     *(buffer + digits) = '\0'; // End of string.
420104
420105     for (b = digits - 1; integer != 0 && b >= 0; b--)
420106     {
420107         remainder = integer % base;
420108         integer = integer / base;
420109
420110         if (remainder <= 9)
420111         {
420112             *(buffer + b) = remainder + '0';
420113         }
420114         else
420115         {
420116             if (uppercase)
420117             {
420118                 *(buffer + b) = remainder - 10 + 'A';
420119             }
420120             else
420121             {
420122                 *(buffer + b) = remainder - 10 + 'a';
420123             }
420124         }
420125     }
420126     return digits;
420127 }
420128 //-----
420129 static size_t
420130 imaxtoa (intmax_t integer, char *buffer, int base,
420131         int uppercase, size_t size)
420132 {
420133     // -----
420134     // Convert a maximum rank integer with sign into a
420135     // string.
420136     // -----
420137
420138     if (integer >= 0)
420139     {
420140         return uimaxtoa (integer, buffer, base,
420141                         uppercase, size);

```

```

420043     }
420044     //
420045     // At this point, there is a negative number, less
420046     // than zero.
420047     //
420048     if (buffer == NULL)
420049     {
420050         return uimaxtoa (-integer, NULL, base, uppercase,
420051             size) + 1;
420052     }
420053
420054     *buffer = '-';           // The minus sign is needed at
420055     // the beginning.
420056     if (size == 1)
420057     {
420058         *(buffer + 1) = '\0';
420059         return 1;
420060     }
420061     else
420062     {
420063         return uimaxtoa (-integer, buffer + 1, base,
420064             uppercase, size - 1) + 1;
420065     }
420066 }
420067
420068 //-----
420069 static size_t
420070 simaxtoa (intmax_t integer, char *buffer, int base,
420071     int uppercase, size_t size)
420072 {
420073     // -----
420074     // Convert a maximum rank integer with sign into a
420075     // string, placing
420076     // the sign also if it is positive.
420077     // -----
420078
420079     if (buffer == NULL && integer >= 0)
420080     {
420081         return uimaxtoa (integer, NULL, base, uppercase,
420082             size) + 1;
420083     }
420084
420085     if (buffer == NULL && integer < 0)
420086     {
420087         return uimaxtoa (-integer, NULL, base, uppercase,
420088             size) + 1;
420089     }
420090     //
420091     // At this point, 'buffer' is different from NULL.
420092     //
420093     if (integer >= 0)
420094     {
420095         *buffer = '+';
420096     }
420097     else
420098     {
420099         *buffer = '-';
420100     }
420101
420102     if (size == 1)
420103     {
420104         *(buffer + 1) = '\0';
420105         return 1;
420106     }
420107
420108     if (integer >= 0)
420109     {
420110         return uimaxtoa (integer, buffer + 1, base,
420111             uppercase, size - 1) + 1;
420112     }
420113     else
420114     {
420115         return uimaxtoa (-integer, buffer + 1, base,
420116             uppercase, size - 1) + 1;
420117     }
420118 }
420119
420120 //-----
420121 static size_t
420122 uimaxtoa_fill (uintmax_t integer, char *buffer,
420123     int base, int uppercase, int width,
420124     int filler, int max)
420125 {
420126     // -----
420127     // Convert a maximum rank integer without sign into
420128     // a string,
420129     // taking care of the alignment.

```

```

420130     // -----
420131
420132     size_t size_i;
420133     size_t size_f;
420134
420135     if (max < 0)
420136         return 0; // «max» deve essere un valore
420137     // positivo.
420138
420139     size_i = uimaxtoa (integer, NULL, base, uppercase, 0);
420140
420141     if (width > 0 && max > 0 && width > max)
420142         width = max;
420143     if (width < 0 && -max < 0 && width < -max)
420144         width = -max;
420145
420146     if (size_i > abs (width))
420147     {
420148         return uimaxtoa (integer, buffer, base,
420149             uppercase, abs (width));
420150     }
420151
420152     if (width == 0 && max > 0)
420153     {
420154         return uimaxtoa (integer, buffer, base,
420155             uppercase, max);
420156     }
420157
420158     if (width == 0)
420159     {
420160         return uimaxtoa (integer, buffer, base,
420161             uppercase, abs (width));
420162     }
420163     //
420164     // size_i <= abs (width).
420165     //
420166     size_f = abs (width) - size_i;
420167
420168     if (width < 0)
420169     {
420170         // Left alignment.
420171         uimaxtoa (integer, buffer, base, uppercase, 0);
420172         memset (buffer + size_i, filler, size_f);
420173     }
420174     else
420175     {
420176         // Right alignment.
420177         memset (buffer, filler, size_f);
420178         uimaxtoa (integer, buffer + size_f, base,
420179             uppercase, 0);
420180     }
420181     *(buffer + abs (width)) = '\0';
420182
420183     return abs (width);
420184 }
420185
420186 //-----
420187 static size_t
420188 imaxtoa_fill (intmax_t integer, char *buffer, int base,
420189     int uppercase, int width, int filler, int max)
420190 {
420191     // -----
420192     // Convert a maximum rank integer with sign into a
420193     // string,
420194     // taking care of the alignment.
420195     // -----
420196
420197     size_t size_i;
420198     size_t size_f;
420199
420200     if (max < 0)
420201         return 0; // 'max' must be a positive value.
420202
420203     size_i = imaxtoa (integer, NULL, base, uppercase, 0);
420204
420205     if (width > 0 && max > 0 && width > max)
420206         width = max;
420207     if (width < 0 && -max < 0 && width < -max)
420208         width = -max;
420209
420210     if (size_i > abs (width))
420211     {
420212         return imaxtoa (integer, buffer, base, uppercase,
420213             abs (width));
420214     }
420215
420216     if (width == 0 && max > 0)

```

```

4201217 {
4201218     return imaxtoa (integer, buffer, base, uppercase,
4201219                   max);
4201220 }
4201221
4201222 if (width == 0)
4201223 {
4201224     return imaxtoa (integer, buffer, base, uppercase,
4201225                   abs (width));
4201226 }
4201227
4201228 // size_i <= abs (width).
4201229
4201230 size_f = abs (width) - size_i;
4201231
4201232 if (width < 0)
4201233 {
4201234     // Left alignment.
4201235     imaxtoa (integer, buffer, base, uppercase, 0);
4201236     memset (buffer + size_i, filler, size_f);
4201237 }
4201238 else
4201239 {
4201240     // Right alignment.
4201241     memset (buffer, filler, size_f);
4201242     imaxtoa (integer, buffer + size_f, base,
4201243           uppercase, 0);
4201244 }
4201245 *(buffer + abs (width)) = '\0';
4201246
4201247 return abs (width);
4201248 }
4201249
4201250 //-----
4201251 static size_t
4201252 simaxtoa_fill (intmax_t integer, char *buffer,
4201253              int base, int uppercase, int width,
4201254              int filler, int max)
4201255 {
4201256     // -----
4201257     // Convert a maximum rank integer with sign into a
4201258     // string,
4201259     // placing the sign also if it is positive and
4201260     // taking care of the
4201261     // alignment.
4201262     // -----
4201263
4201264     size_t size_i;
4201265     size_t size_f;
4201266
4201267     if (max < 0)
4201268         return 0; // 'max' must be a positive value.
4201269
4201270     size_i = simaxtoa (integer, NULL, base, uppercase, 0);
4201271
4201272     if (width > 0 && max > 0 && width > max)
4201273         width = max;
4201274     if (width < 0 && -max < 0 && width < -max)
4201275         width = -max;
4201276
4201277     if (size_i > abs (width))
4201278     {
4201279         return simaxtoa (integer, buffer, base,
4201280                       uppercase, abs (width));
4201281     }
4201282
4201283     if (width == 0 && max > 0)
4201284     {
4201285         return simaxtoa (integer, buffer, base,
4201286                       uppercase, max);
4201287     }
4201288
4201289     if (width == 0)
4201290     {
4201291         return simaxtoa (integer, buffer, base,
4201292                       uppercase, abs (width));
4201293     }
4201294     //
4201295     // size_i <= abs (width).
4201296     //
4201297     size_f = abs (width) - size_i;
4201298
4201299     if (width < 0)
4201300     {
4201301         // Left alignment.
4201302         simaxtoa (integer, buffer, base, uppercase, 0);
4201303         memset (buffer + size_i, filler, size_f);

```

```

4201304     }
4201305     else
4201306     {
4201307         // Right alignment.
4201308         memset (buffer, filler, size_f);
4201309         simaxtoa (integer, buffer + size_f, base,
4201310               uppercase, 0);
4201311     }
4201312     *(buffer + abs (width)) = '\0';
4201313
4201314     return abs (width);
4201315 }
4201316
4201317 //-----
4201318 static size_t
4201319 strtost_r_fill (char *string, char *buffer, int width,
4201320               int filler, int max)
4201321 {
4201322     // -----
4201323     // Transfer a string with care for the alignment.
4201324     // -----
4201325
4201326     size_t size_s;
4201327     size_t size_f;
4201328
4201329     if (max < 0)
4201330         return 0; // 'max' must be a positive value.
4201331
4201332     size_s = strlen (string);
4201333
4201334     if (width > 0 && max > 0 && width > max)
4201335         width = max;
4201336     if (width < 0 && -max < 0 && width < -max)
4201337         width = -max;
4201338
4201339     if (width != 0 && size_s > abs (width))
4201340     {
4201341         memcpy (buffer, string, abs (width));
4201342         buffer[width] = '\0';
4201343         return width;
4201344     }
4201345
4201346     if (width == 0 && max > 0 && size_s > max)
4201347     {
4201348         memcpy (buffer, string, max);
4201349         buffer[max] = '\0';
4201350         return max;
4201351     }
4201352
4201353     if (width == 0 && max > 0 && size_s < max)
4201354     {
4201355         memcpy (buffer, string, size_s);
4201356         buffer[size_s] = '\0';
4201357         return size_s;
4201358     }
4201359     //
4201360     // width != 0
4201361     // size_s <= abs (width)
4201362     //
4201363     size_f = abs (width) - size_s;
4201364
4201365     if (width < 0)
4201366     {
4201367         // Right alignment.
4201368         memset (buffer, filler, size_f);
4201369         strncpy (buffer + size_f, string, size_s);
4201370     }
4201371     else
4201372     {
4201373         // Left alignment.
4201374         strncpy (buffer, string, size_s);
4201375         memset (buffer + size_s, filler, size_f);
4201376     }
4201377     *(buffer + abs (width)) = '\0';
4201378
4201379     return abs (width);
4201380 }

```

95.18.43 lib/stdio/vsprintf.c

Si veda la sezione 88.137.

```

4210001 #include <stdio.h>
4210002 //-----
4210003 int
4210004 vsprintf (char *restrict string,
4210005          const char *restrict format, va_list arg)

```



```

421006 {
421007     return (vsnprintf (string, BUFSIZ, format, arg));
421008 }

```

## 95.18.44 lib/stdio/vsscanf.c

« Si veda la sezione 88.138.

```

422001 #include <stdio.h>
422002
422003 //-----
422004 int vfscanf (FILE * restrict fp, const char *string,
422005             const char *restrict format, va_list ap);
422006 //-----
422007 int
422008 vsscanf (const char *string,
422009         const char *restrict format, va_list ap)
422010 {
422011     return (vfscanf (NULL, string, format, ap));
422012 }
422013 //-----
422014

```

## 95.19 os32: «lib/stdlib.h»

« Si veda la sezione 91.3.

```

423001 #ifndef _STDLIB_H
423002 #define _STDLIB_H    1
423003 //-----
423004 #include <size_t.h>
423005 #include <wchar_t.h>
423006 #include <NULL.h>
423007 #include <limits.h>
423008 #include <restrict.h>
423009 #include <stdint.h>
423010 //-----
423011 typedef struct
423012 {
423013     int quot;
423014     int rem;
423015 } div_t;
423016 //-----
423017 typedef struct
423018 {
423019     long int quot;
423020     long int rem;
423021 } ldiv_t;
423022 //-----
423023 typedef struct
423024 {
423025     long long int quot;
423026     long long int rem;
423027 } lldiv_t;
423028 //-----
423029 typedef void (*atexit_t) (void);    // Non standard.
423030 // [1]
423031 //
423032 // [1] The type 'atexit_t' is a pointer to a function
423033 // for the "at exit" procedure, with no parameters
423034 // and returning void. With the declaration of type
423035 // 'atexit_t', the function prototype of 'atexit()'
423036 // is easier to declare and to understand. Original
423037 // declaration is:
423038 //
423039 //     int atexit (void (*function) (void));
423040 //
423041 //-----
423042 typedef struct
423043 {
423044     uintptr_t allocated:1, filler:1, next:30;
423045 } _alloc_head_t;    // Non standard [2]
423046 //
423047 // [2] This is used for the 'malloc()' management, as
423048 // the pointer to the following element of memory,
423049 // that might be free or allocated.
423050 //
423051 // La dimensione di «uintptr_t» condiziona la struttura
423052 // «mm_head_t» e la dimensione delle unità minime di
423053 // memoria allocata. «uintptr_t» è da 32 bit, così
423054 // l'immagine del kernel è allineata a blocchi da
423055 // 32 bit e così deve essere anche per gli altri
423056 // blocchi di memoria.
423057 // Essendo i blocchi di memoria multipli di 32 bit, gli
423058 // indirizzi sono sempre multipli di 4 (4 byte);
423059 // pertanto, servono solo 30 bit per rappresentare
423060 // l'indirizzo, che poi viene ottenuto moltiplicandolo

```

```

423061 // per quattro. Di conseguenza, il bit meno
423062 // significativo viene usato per annotare se il blocco
423063 // di memoria è libero e il bit successivo non viene
423064 // usato. Questo meccanismo potrebbe essere usato anche
423065 // con un indirizzamento a 16 bit, dove servirebbero 15
423066 // bit per indirizzi multipli di due byte.
423067 //-----
423068 #define EXIT_FAILURE    1
423069 #define EXIT_SUCCESS    0
423070 #define RAND_MAX        INT_MAX
423071 #define MB_CUR_MAX      ((size_t) MB_LEN_MAX)
423072 //-----
423073 void _Exit (int status);
423074 void abort (void);
423075 int abs (int j);
423076 int atexit (atexit_t function);
423077 int atoi (const char *string);
423078 long int atol (const char *string);
423079 #define calloc(b, s) (malloc ((b) * (s)))
423080 div_t div (int numer, int denom);
423081 void exit (int status);
423082 void free (void *ptr);
423083 char *getenv (const char *name);
423084 long int labs (long int j);
423085 long long int llabs (long long int j);
423086 ldiv_t ldiv (long int numer, long int denom);
423087 lldiv_t lldiv (long long int numer, long long int denom);
423088 void *malloc (size_t size);
423089 int putenv (const char *string);
423090 void qsort (void *base, size_t nmemb, size_t size,
423091           int (*compare) (const void *, const void *));
423092 int rand (void);
423093 void *realloc (void *ptr, size_t size);
423094 int setenv (const char *name, const char *value,
423095           int overwrite);
423096 void srand (unsigned int seed);
423097 long int strtol (const char *restrict string,
423098                char **restrict endptr, int base);
423099 unsigned long int strtoul (const char *restrict string,
423100                           char **restrict endptr,
423101                           int base);
423102 //int          system (const char *string);
423103 int unsetenv (const char *name);
423104 //-----
423105 #endif

```

95.19.1	lib/stdlib/_Exit.c	846
95.19.2	lib/stdlib/abort.c	846
95.19.3	lib/stdlib/abs.c	846
95.19.4	lib/stdlib/atexit.c	847
95.19.5	lib/stdlib/atoi.c	847
95.19.6	lib/stdlib/atol.c	848
95.19.7	lib/stdlib/div.c	848
95.19.8	lib/stdlib/environment.c	848
95.19.9	lib/stdlib/exit.c	849
95.19.10	lib/stdlib/getenv.c	850
95.19.11	lib/stdlib/labs.c	851
95.19.12	lib/stdlib/ldiv.c	851
95.19.13	lib/stdlib/llabs.c	851
95.19.14	lib/stdlib/lldiv.c	851
95.19.15	lib/stdlib/putenv.c	851
95.19.16	lib/stdlib/qsort.c	853
95.19.17	lib/stdlib/rand.c	854
95.19.18	lib/stdlib/setenv.c	855
95.19.19	lib/stdlib/strtoul.c	856
95.19.20	lib/stdlib/strtoul.c	859
95.19.21	lib/stdlib/unsetenv.c	859
95.19.22	lib/stdlib/_alloc/_alloc_list.c	860
95.19.23	lib/stdlib/_alloc/free.c	861

95.19.24	lib/stdlib_alloc/malloc.c	862
95.19.25	lib/stdlib_alloc/realloc.c	865

## 95.19.1 lib/stdlib/\_Exit.c

« Si veda la sezione 87.2.

```

4240001 #include <stdlib.h>
4240002 #include <sys/os32.h>
4240003 //-----
4240004 void
4240005 _Exit (int status)
4240006 {
4240007     sysmsg_exit_t msg;
4240008     //
4240009     // Only the low eight bit are returned.
4240010     //
4240011     msg.status = (status & 0xFF);
4240012     //
4240013     //
4240014     //
4240015     sys (SYS_EXIT, &msg, (sizeof msg));
4240016     //
4240017     // Should not return from system call, but if it
4240018     // does, loop
4240019     // forever:
4240020     //
4240021     while (1);
4240022 }

```

## 95.19.2 lib/stdlib/abort.c

« Si veda la sezione 88.2.

```

4250001 #include <stdlib.h>
4250002 #include <sys/types.h>
4250003 #include <signal.h>
4250004 #include <unistd.h>
4250005 //-----
4250006 void
4250007 abort (void)
4250008 {
4250009     pid_t pid;
4250010     sighandler_t sig_previous;
4250011     //
4250012     // Set 'SIGABRT' to a default action.
4250013     //
4250014     sig_previous = signal (SIGABRT, SIG_DFL);
4250015     //
4250016     // If the previous action was something different
4250017     // than symbolic
4250018     // ones, configure again the previous action.
4250019     //
4250020     if (sig_previous != SIG_DFL &&
4250021         sig_previous != SIG_IGN && sig_previous != SIG_ERR)
4250022     {
4250023         signal (SIGABRT, sig_previous);
4250024     }
4250025     //
4250026     // Get current process ID and sent the signal.
4250027     //
4250028     pid = getpid ();
4250029     kill (pid, SIGABRT);
4250030     //
4250031     // Second chance
4250032     //
4250033     for (;;)
4250034     {
4250035         signal (SIGABRT, SIG_DFL);
4250036         pid = getpid ();
4250037         kill (pid, SIGABRT);
4250038     }
4250039 }

```

## 95.19.3 lib/stdlib/abs.c

« Si veda la sezione 88.3.

```

4260001 #include <stdlib.h>
4260002 //-----
4260003 int
4260004 abs (int j)
4260005 {
4260006     if (j < 0)
4260007     {
4260008         return -j;
4260009     }

```

```

4260010     else
4260011     {
4260012         return j;
4260013     }
4260014 }

```

## 95.19.4 lib/stdlib/atexit.c

« Si veda la sezione 88.7.

```

4270001 #include <stdlib.h>
4270002 //-----
4270003 atexit_t _atexit_table[ATEXIT_MAX];
4270004 //-----
4270005 void
4270006 _atexit_setup (void)
4270007 {
4270008     int a;
4270009     //
4270010     for (a = 0; a < ATEXTIT_MAX; a++)
4270011     {
4270012         _atexit_table[a] = NULL;
4270013     }
4270014 }
4270015 //-----
4270016 int
4270017 atexit (atexit_t function)
4270018 {
4270019     int a;
4270020     //
4270021     if (function == NULL)
4270022     {
4270023         return (-1);
4270024     }
4270025     //
4270026     for (a = 0; a < ATEXTIT_MAX; a++)
4270027     {
4270028         if (_atexit_table[a] == NULL)
4270029         {
4270030             _atexit_table[a] = function;
4270031             return (0);
4270032         }
4270033     }
4270034     //
4270035     return (-1);
4270036 }
4270037 }

```

## 95.19.5 lib/stdlib/atoi.c

« Si veda la sezione 88.8.

```

4280001 #include <stdlib.h>
4280002 #include <ctype.h>
4280003 //-----
4280004 int
4280005 atoi (const char *string)
4280006 {
4280007     int i;
4280008     int sign = +1;
4280009     int number;
4280010     //
4280011     for (i = 0; isspace (string[i]); i++)
4280012     {
4280013         ;
4280014     }
4280015     //
4280016     if (string[i] == '+')
4280017     {
4280018         sign = +1;
4280019         i++;
4280020     }
4280021     else if (string[i] == '-')
4280022     {
4280023         sign = -1;
4280024         i++;
4280025     }
4280026     //
4280027     for (number = 0; isdigit (string[i]); i++)
4280028     {
4280029         number *= 10;
4280030         number += (string[i] - '0');
4280031     }
4280032     //
4280033     number *= sign;
4280034     //
4280035     return number;

```

```
428036 }
```

## 95.19.6 lib/stdlib/atol.c

« Si veda la sezione 88.8.

```
428001 #include <stdlib.h>
428002 #include <ctype.h>
428003 //-----
428004 long int
428005 atol (const char *string)
428006 {
428007     int i;
428008     int sign = +1;
428009     long int number;
428010     //
428011     for (i = 0; isspace (string[i]); i++)
428012     {
428013         ;
428014     }
428015     //
428016     if (string[i] == '+')
428017     {
428018         sign = +1;
428019         i++;
428020     }
428021     else if (string[i] == '-')
428022     {
428023         sign = -1;
428024         i++;
428025     }
428026     //
428027     for (number = 0; isdigit (string[i]); i++)
428028     {
428029         number *= 10;
428030         number += (string[i] - '0');
428031     }
428032     //
428033     number *= sign;
428034     //
428035     return number;
428036 }
```

## 95.19.7 lib/stdlib/div.c

« Si veda la sezione 88.17.

```
430001 #include <stdlib.h>
430002 //-----
430003 div_t
430004 div (int numer, int denom)
430005 {
430006     div_t d;
430007     d.quot = numer / denom;
430008     d.rem = numer % denom;
430009     return d;
430010 }
```

## 95.19.8 lib/stdlib/environment.c

« Si veda la sezione 91.1.

```
431001 #include <stdlib.h>
431002 #include <string.h>
431003 //-----
431004 // This file contains a non standard definition,
431005 // related to the environment handling.
431006 //
431007 // The file 'crt0.s', before calling the main function,
431008 // calls the function '_environment_setup()', that is
431009 // responsible for initializing the array
431010 // '_environment_table[[]]' and for copying the content
431011 // of the environment, as it comes from the 'exec()'
431012 // system call.
431013 //
431014 // The pointers to the environment strings organised
431015 // inside the array '_environment_table[[]]', are also
431016 // copied inside the array of pointers
431017 // '_environment[]'.
431018 //
431019 // After all that is done, inside 'crt0.s', the pointer
431020 // to 'environment[]' is copied to the traditional
431021 // variable 'environ' and also to the previous value of
431022 // the pointer variable 'envp'.
431023 //
431024 // This way, applications will get the environment, but
431025 // organised inside the table '_environment_table[[]]'.
```

```
431026 // So, functions like 'getenv()' and 'setenv()' do know
431027 // where to look for.
431028 //
431029 // It is useful to notice that there is no prototype
431030 // and no extern declaration inside the file
431031 // <stdlib.h>, about this function and these arrays,
431032 // because applications do not have to know about it.
431033 //
431034 // Please notice that 'environ' could be just the same
431035 // as '_environment' here, but the common use puts
431036 // 'environ' inside <unistd.h>, although for this
431037 // implementation it should be better placed inside
431038 // <stdlib.h>.
431039 //
431040 //-----
431041 char _environment_table[ARG_MAX / 32][ARG_MAX / 16];
431042 char *_environment[ARG_MAX / 32 + 1];
431043 //-----
431044 void
431045 _environment_setup (char *envp[])
431046 {
431047     int e;
431048     int s;
431049     //
431050     // Reset the '_environment_table[[]]' array.
431051     //
431052     for (e = 0; e < ARG_MAX / 32; e++)
431053     {
431054         for (s = 0; s < ARG_MAX / 16; s++)
431055         {
431056             _environment_table[e][s] = 0;
431057         }
431058     }
431059     //
431060     // Set the '_environment[[]]' pointers. The final
431061     // extra element must
431062     // be a NULL pointer.
431063     //
431064     for (e = 0; e < ARG_MAX / 32; e++)
431065     {
431066         _environment[e] = _environment_table[e];
431067     }
431068     _environment[ARG_MAX / 32] = NULL;
431069     //
431070     // Copy the environment inside the array, but only
431071     // if 'envp' is
431072     // not NULL.
431073     //
431074     if (envp != NULL)
431075     {
431076         for (e = 0; envp[e] != NULL && e < ARG_MAX / 32; e++)
431077         {
431078             strncpy (_environment_table[e], envp[e],
431079                 (ARG_MAX / 16) - 1);
431080         }
431081     }
431082 }
```

## 95.19.9 lib/stdlib/exit.c

« Si veda la sezione 88.7.

```
432001 #include <stdlib.h>
432002 #include <stdio.h>
432003 //-----
432004 extern atexit_t _atexit_table[];
432005 //-----
432006 void
432007 exit (int status)
432008 {
432009     int a;
432010     //
432011     // The "at exit" functions must be called in reverse
432012     // order.
432013     //
432014     for (a = (ATEXIT_MAX - 1); a >= 0; a--)
432015     {
432016         if (_atexit_table[a] != NULL)
432017         {
432018             (*_atexit_table[a]) ();
432019         }
432020     }
432021     //
432022     // Now: really exit.
432023     //
432024     _Exit (status);
432025     //
```

```

432026 // Should not return from system call, but if it
432027 // does, loop
432028 // forever:
432029 //
432030 while (1);
432031 }

```

## 95.19.10 lib/stdlib/getenv.c

« Si veda la sezione 88.52.

```

433001 #include <stdlib.h>
433002 #include <string.h>
433003 //-----
433004 extern char *_environment[];
433005 //-----
433006 char *
433007 getenv (const char *name)
433008 {
433009     int e; // First index: environment table
433010 // items.
433011 int f; // Second index: environment string
433012 // scan.
433013 char *value; // Pointer to the environment value
433014 // found.
433015 //
433016 // Check if the input is valid. No error is
433017 // reported.
433018 //
433019 if (name == NULL || strlen (name) == 0)
433020 {
433021     return (NULL);
433022 }
433023 //
433024 // Scan the environment table items, with index 'e'.
433025 // The pointer
433026 // 'value' is initialized to NULL. If the pointer
433027 // 'value' gets a
433028 // valid pointer, the environment variable was found
433029 // and a
433030 // pointer to the beginning of its value is
433031 // available.
433032 //
433033 for (value = NULL, e = 0; e < ARG_MAX / 32; e++)
433034 {
433035     //
433036     // Scan the string of the environment item, with
433037     // index 'f'.
433038     // The scan continue until 'name[f]' and
433039     // '_environment[e][f]'
433040     // are equal.
433041     //
433042     for (f = 0;
433043          f < ARG_MAX / 16 - 1
433044          && name[f] == _environment[e][f]; f++)
433045     {
433046         // Just scan.
433047     }
433048     //
433049     // At this point, 'name[f]' and
433050     // '_environment[e][f]' are
433051     // different: if 'name[f]' is zero the name
433052     // string is
433053     // terminated; if '_environment[e][f]' is also
433054     // equal to '=',
433055     // the environment item is corresponding to the
433056     // requested name.
433057     //
433058     if (name[f] == 0 && _environment[e][f] == '=')
433059     {
433060         //
433061         // The pointer to the beginning of the
433062         // environment value is
433063         // calculated, and the external loop exit.
433064         //
433065         value = &_environment[e][f + 1];
433066         break;
433067     }
433068 }
433069 //
433070 // The 'value' is returned: if it is still NULL,
433071 // then, no
433072 // environment variable with the requested name was
433073 // found.
433074 //
433075 return (value);
433076 }

```

## 95.19.11 lib/stdlib/labs.c

« Si veda la sezione 88.3.

```

434001 #include <stdlib.h>
434002 //-----
434003 long int
434004 labs (long int j)
434005 {
434006     if (j < 0)
434007     {
434008         return -j;
434009     }
434010     else
434011     {
434012         return j;
434013     }
434014 }

```

## 95.19.12 lib/stdlib/ldiv.c

« Si veda la sezione 88.17.

```

435001 #include <stdlib.h>
435002 //-----
435003 ldiv_t
435004 ldiv (long int numer, long int denom)
435005 {
435006     ldiv_t d;
435007     d.quot = numer / denom;
435008     d.rem = numer % denom;
435009     return d;
435010 }

```

## 95.19.13 lib/stdlib/llabs.c

« Si veda la sezione 88.3.

```

436001 #include <stdlib.h>
436002 //-----
436003 long long int
436004 llabs (long long int j)
436005 {
436006     if (j < 0)
436007     {
436008         return -j;
436009     }
436010     else
436011     {
436012         return j;
436013     }
436014 }

```

## 95.19.14 lib/stdlib/lldiv.c

« Si veda la sezione 88.17.

```

437001 #include <stdlib.h>
437002 //-----
437003 lldiv_t
437004 lldiv (long long int numer, long long int denom)
437005 {
437006     lldiv_t d;
437007     d.quot = numer / denom;
437008     d.rem = numer % denom;
437009     return d;
437010 }

```

## 95.19.15 lib/stdlib/putenv.c

« Si veda la sezione 88.94.

```

438001 #include <stdlib.h>
438002 #include <string.h>
438003 #include <errno.h>
438004 //-----
438005 extern char *_environment[];
438006 //-----
438007 int
438008 putenv (const char *string)
438009 {
438010     int e; // First index: environment table
438011 // items.
438012 int f; // Second index: environment string
438013 // scan.
438014 //
438015 // Check if the input is empty. No error is
438016 // reported.

```

```

438017 //
438018 if (string == NULL || strlen (string) == 0)
438019 {
438020     return (0);
438021 }
438022 //
438023 // Check if the input is valid: there must be a '='
438024 // sign.
438025 // Error here is reported.
438026 //
438027 if (strchr (string, '=') == NULL)
438028 {
438029     errset (EINVAL); // Invalid argument.
438030     return (-1);
438031 }
438032 //
438033 // Scan the environment table items, with index 'e'.
438034 // The intent is
438035 // to find a previous environment variable with the
438036 // same name.
438037 //
438038 for (e = 0; e < ARG_MAX / 32; e++)
438039 {
438040     //
438041     // Scan the string of the environment item, with
438042     // index 'f'.
438043     // The scan continue until 'string[f]' and
438044     // '_environment[e][f]'
438045     // are equal.
438046     //
438047     for (f = 0;
438048          f < ARG_MAX / 16 - 1
438049          && string[f] == _environment[e][f]; f++)
438050     {
438051         ; // Just scan.
438052     }
438053     //
438054     // At this point, 'string[f-1]' and
438055     // '_environment[e][f-1]'
438056     // should contain '='. If it is so, the
438057     // environment is replaced.
438058     //
438059     if (string[f - 1] == '='
438060         && _environment[e][f - 1] == '=')
438061     {
438062         //
438063         // The environment item was found: now
438064         // replace the pointer.
438065         //
438066         _environment[e] = (char *) string;
438067         //
438068         // Return.
438069         //
438070         return (0);
438071     }
438072 }
438073 //
438074 // The item was not found. Scan again for a free
438075 // slot.
438076 //
438077 for (e = 0; e < ARG_MAX / 32; e++)
438078 {
438079     if (_environment[e] == NULL
438080         || _environment[e][0] == 0)
438081     {
438082         //
438083         // An empty item was found and the pointer
438084         // will be
438085         // replaced.
438086         //
438087         _environment[e] = (char *) string;
438088         //
438089         // Return.
438090         //
438091         return (0);
438092     }
438093 }
438094 //
438095 // Sorry: the empty slot was not found!
438096 //
438097 errset (ENOMEM); // Not enough space.
438098 return (-1);
438099 }

```

95.19.16 lib/stdlib/qsort.c

Si veda la sezione 88.96.

```

490001 #include <stdlib.h>
490002 #include <string.h>
490003 #include <errno.h>
490004 //-----
490005 static int part (char *array, size_t size, int a,
490006                int z, int (*compare) (const void *,
490007                                       const void *));
490008 static void sort (char *array, size_t size, int a,
490009                 int z, int (*compare) (const void *,
490010                                       const void *));
490011 //-----
490012 void
490013 qsort (void *base, size_t nmemb, size_t size,
490014       int (*compare) (const void *, const void *))
490015 {
490016     if (size <= 1)
490017     {
490018         //
490019         // There is nothing to sort!
490020         //
490021         return;
490022     }
490023     else
490024     {
490025         sort ((char *) base, size, 0, (int) (nmemb - 1),
490026             compare);
490027     }
490028 }
490029 //-----
490030 static void
490031 sort (char *array, size_t size, int a, int z,
490032      int (*compare) (const void *, const void *))
490033 {
490034     int loc;
490035     //
490036     // if (z > a)
490037     {
490038         {
490039             loc = part (array, size, a, z, compare);
490040             if (loc >= 0)
490041             {
490042                 sort (array, size, a, loc - 1, compare);
490043                 sort (array, size, loc + 1, z, compare);
490044             }
490045         }
490046     }
490047 }
490048 //-----
490049 static int
490050 part (char *array, size_t size, int a, int z,
490051      int (*compare) (const void *, const void *))
490052 {
490053     int i;
490054     int loc;
490055     char *swap;
490056     //
490057     if (z <= a)
490058     {
490059         errset (EUNKNOWN); // Should never
490060                             // happen.
490061         return (-1);
490062     }
490063     //
490064     // Index 'i' after the first element; index 'loc' at
490065     // the last
490066     // position.
490067     //
490068     i = a + 1;
490069     loc = z;
490070     //
490071     // Prepare space in memory for element swap.
490072     //
490073     swap = malloc (size);
490074     if (swap == NULL)
490075     {
490076         errset (ENOMEM);
490077         return (-1);
490078     }
490079     //
490080     // Loop as long as index 'loc' is higher than index
490081     // 'i'.
490082     // When index 'loc' is less or equal to index 'i',
490083     // then, index 'loc' is the right position for the
490084     // first element of the current piece of array.
490085     //

```

```

439086 for (;;)
439087 {
439088     //
439089     // Index 'i' goes up...
439090     //
439091     for (; i < loc; i++)
439092     {
439093         if (compare
439094             (&array[i * size], &array[a * size]) > 0)
439095         {
439096             break;
439097         }
439098     }
439099     //
439100     // Index 'loc' goes down...
439101     //
439102     for (; loc--;)
439103     {
439104         if (compare
439105             (&array[loc * size], &array[a * size]) <= 0)
439106         {
439107             break;
439108         }
439109     }
439110     //
439111     // Swap elements related to index 'i' and 'loc'.
439112     //
439113     if (loc <= i)
439114     {
439115         //
439116         // The array is completely scanned.
439117         //
439118         break;
439119     }
439120     else
439121     {
439122         memcpy (swap, &array[loc * size], size);
439123         memcpy (&array[loc * size], &array[i * size],
439124             size);
439125         memcpy (&array[i * size], swap, size);
439126     }
439127 }
439128 //
439129 // Swap the first element with the one related to
439130 // the
439131 // index 'loc'.
439132 //
439133 memcpy (swap, &array[loc * size], size);
439134 memcpy (&array[loc * size], &array[a * size], size);
439135 memcpy (&array[a * size], swap, size);
439136 //
439137 // Free the swap memory.
439138 //
439139 free (swap);
439140 //
439141 // Return the index 'loc'.
439142 //
439143 return (loc);
439144 }

```

## 95.19.17 lib/stdlib/rand.c

« Si veda la sezione 88.97.

```

440001 #include <stdlib.h>
440002 //-----
440003 static unsigned int _srand = 1; // The '_srand' rank
440004 // must be at least
440005 // 'unsigned int' and
440006 // must be able to
440007 // represent the value
440008 // 'RAND_MAX'.
440009 //-----
440010 int
440011 rand (void)
440012 {
440013     _srand = _srand * 12345 + 123;
440014     return _srand % ((unsigned int) RAND_MAX + 1);
440015 }
440016 //-----
440017 void
440018 srand (unsigned int seed)
440019 {
440020     _srand = seed;
440021 }
440022 }

```

## 95.19.18 lib/stdlib/setenv.c

« Si veda la sezione 88.104.

```

441001 #include <stdlib.h>
441002 #include <string.h>
441003 #include <errno.h>
441004 //-----
441005 extern char *_environment[];
441006 extern char *_environment_table[];
441007 //-----
441008 int
441009 setenv (const char *name, const char *value, int overwrite)
441010 {
441011     int e; // First index: environment table
441012 // items.
441013     int f; // Second index: environment string
441014 // scan.
441015 //
441016 // Check if the input is empty. No error is
441017 // reported.
441018 //
441019 if (name == NULL || strlen (name) == 0)
441020 {
441021     return (0);
441022 }
441023 //
441024 // Check if the input is valid: error here is
441025 // reported.
441026 //
441027 if (strchr (name, '=') != NULL)
441028 {
441029     errset (EINVAL); // Invalid argument.
441030     return (-1);
441031 }
441032 //
441033 // Check if the input is too big.
441034 //
441035 if ((strlen (name) + strlen (value) + 2) > ARG_MAX / 16)
441036 {
441037     //
441038     // The environment to be saved is bigger than
441039     // the
441040     // available string size, inside
441041     // '_environment_table[]'.
441042     //
441043     errset (ENOMEM); // Not enough space.
441044     return (-1);
441045 }
441046 //
441047 // Scan the environment table items, with index 'e'.
441048 // The intent is
441049 // to find a previous environment variable with the
441050 // same name.
441051 //
441052 for (e = 0; e < ARG_MAX / 32; e++)
441053 {
441054     //
441055     // Scan the string of the environment item, with
441056     // index 'f'.
441057     // The scan continue until 'name[f]' and
441058     // '_environment[e][f]'
441059     // are equal.
441060     //
441061     for (f = 0;
441062          f < ARG_MAX / 16 - 1
441063          && name[f] == _environment[e][f]; f++)
441064     {
441065         // Just scan.
441066     }
441067 //
441068 // At this point, 'name[f]' and
441069 // '_environment[e][f]' are
441070 // different: if 'name[f]' is zero the name
441071 // string is
441072 // terminated; if '_environment[e][f]' is also
441073 // equal to '=',
441074 // the environment item is corresponding to the
441075 // requested name.
441076 //
441077 if (name[f] == 0 && _environment[e][f] == '=')
441078 {
441079     //
441080     // The environment item was found; if it can
441081     // be overwritten,
441082     // the write is done.
441083     //
441084     if (overwrite)
441085     {

```

```

441086 //
441087 // To be able to handle both 'setenv()'
441088 // and 'putenv()',
441089 // before removing the item, it is fixed
441090 // the pointer to
441091 // the global environment table.
441092 //
441093 // _environment[e] = _environment_table[e];
441094 //
441095 // Now copy the new environment. The
441096 // string size was
441097 // already checked.
441098 //
441099 strcpy (_environment[e], name);
441100 strcat (_environment[e], "=");
441101 strcat (_environment[e], value);
441102 //
441103 // Return.
441104 //
441105 return (0);
441106 }
441107 //
441108 // Cannot overwrite!
441109 //
441110 errset (EUNKNOWN);
441111 return (-1);
441112 }
441113 }
441114 //
441115 // The item was not found. Scan again for a free
441116 // slot.
441117 //
441118 for (e = 0; e < ARG_MAX / 32; e++)
441119 {
441120     if (_environment[e] == NULL
441121         || _environment[e][0] == 0)
441122     {
441123         //
441124         // An empty item was found. To be able to
441125         // handle both
441126         // 'setenv()' and 'putenv()', it is fixed
441127         // the pointer to
441128         // the global environment table.
441129         //
441130         _environment[e] = _environment_table[e];
441131         //
441132         // Now copy the new environment. The string
441133         // size was
441134         // already checked.
441135         //
441136         strcpy (_environment[e], name);
441137         strcat (_environment[e], "=");
441138         strcat (_environment[e], value);
441139         //
441140         // Return.
441141         //
441142         return (0);
441143     }
441144 }
441145 //
441146 // Sorry: the empty slot was not found!
441147 //
441148 errset (ENOMEM); // Not enough space.
441149 return (-1);
441150 }

```

## 95.19.19 lib/stdlib/strtol.c

« Si veda la sezione 88.130.

```

442001 #include <stdlib.h>
442002 #include <ctype.h>
442003 #include <errno.h>
442004 #include <limits.h>
442005 #include <stdbool.h>
442006 //-----
442007 #define isoctal(C) ((int) (C >= '0' && C <= '7'))
442008 //-----
442009 long int
442010 strtol (const char *restrict string,
442011         char **restrict endptr, int base)
442012 {
442013     int i;
442014     int sign = +1;
442015     long int number;
442016     long int previous;
442017     int digit;

```

```

442018 //
442019 bool flag_prefix_oct = 0;
442020 bool flag_prefix_exa = 0;
442021 bool flag_prefix_dec = 0;
442022 //
442023 // Check base and string.
442024 //
442025 // With base 1 cannot do anything.
442026 //
442027 if (base < 0 || base > 36 || base == 1
442028     || string == NULL || string[0] == 0)
442029 {
442030     if (endptr != NULL)
442031         *endptr = (char *) string;
442032     errset (EINVAL); // Invalid argument.
442033     return ((long int) 0);
442034 }
442035 //
442036 // Eat initial spaces.
442037 //
442038 for (i = 0; isspace (string[i]); i++)
442039 {
442040     ;
442041 }
442042 //
442043 // Check sign.
442044 //
442045 if (string[i] == '+')
442046 {
442047     sign = +1;
442048     i++;
442049 }
442050 else if (string[i] == '-')
442051 {
442052     sign = -1;
442053     i++;
442054 }
442055 //
442056 // Check for prefix.
442057 //
442058 if (string[i] == '0')
442059 {
442060     if (string[i + 1] == 'x' || string[i + 1] == 'X')
442061     {
442062         flag_prefix_exa = 1;
442063     }
442064     else if (isoctal (string[i + 1]))
442065     {
442066         flag_prefix_oct = 1;
442067     }
442068     else
442069     {
442070         flag_prefix_dec = 1;
442071     }
442072 }
442073 else if (isdigit (string[i]))
442074 {
442075     flag_prefix_dec = 1;
442076 }
442077 //
442078 // Check compatibility with requested base.
442079 //
442080 if (flag_prefix_exa)
442081 {
442082     //
442083     // At the moment, there is a zero and a 'x'.
442084     // Might be
442085     // hexadecimal, or might be a number base 33 or
442086     // more.
442087     //
442088     if (base == 0)
442089     {
442090         base = 16;
442091     }
442092     else if (base == 16)
442093     {
442094         ; // Ok.
442095     }
442096     else if (base >= 33)
442097     {
442098         ; // Ok.
442099     }
442100     else
442101     {
442102         //
442103         // Incompatible sequence: only the initial
442104         // zero is reported.

```

```

4420105 //
4420106     if (endptr != NULL)
4420107         *endptr = (char *) &string[i + 1];
4420108     return ((long int) 0);
4420109 }
4420110 //
4420111 // Move on, after the '0x' prefix.
4420112 //
4420113     i += 2;
4420114 }
4420115 //
4420116 if (flag_prefix_oct)
4420117 {
4420118     //
4420119     // There is a zero and a digit.
4420120     //
4420121     if (base == 0)
4420122     {
4420123         base = 8;
4420124     }
4420125     //
4420126     // Move on, after the '0' prefix.
4420127     //
4420128     i += 1;
4420129 }
4420130 //
4420131 if (flag_prefix_dec)
4420132 {
4420133     if (base == 0)
4420134     {
4420135         base = 10;
4420136     }
4420137 }
4420138 //
4420139 // Scan the string.
4420140 //
4420141 for (number = 0; string[i] != 0; i++)
4420142 {
4420143     if (string[i] >= '0' && string[i] <= '9')
4420144     {
4420145         digit = string[i] - '0';
4420146     }
4420147     else if (string[i] >= 'A' && string[i] <= 'Z')
4420148     {
4420149         digit = string[i] - 'A' + 10;
4420150     }
4420151     else if (string[i] >= 'a' && string[i] <= 'z')
4420152     {
4420153         digit = string[i] - 'a' + 10;
4420154     }
4420155     else
4420156     {
4420157         //
4420158         // This is an out of range digit.
4420159         //
4420160         digit = 999;
4420161     }
4420162     //
4420163     // Give a sign to the digit.
4420164     //
4420165     digit *= sign;
4420166     //
4420167     // Compare with the base.
4420168     //
4420169     if (base > (digit * sign))
4420170     {
4420171         //
4420172         // Check if the current digit can be safely
4420173         // computed.
4420174         //
4420175         previous = number;
4420176         number *= base;
4420177         number += digit;
4420178         if (number / base != previous)
4420179         {
4420180             //
4420181             // Out of range.
4420182             //
4420183             if (endptr != NULL)
4420184                 *endptr = (char *) &string[i + 1];
4420185             errset (ERANGE); // Result too large.
4420186             if (sign > 0)
4420187             {
4420188                 return (LONG_MAX);
4420189             }
4420190             else
4420191             {

```

```

4420192         return (LONG_MIN);
4420193     }
4420194 }
4420195 }
4420196     else
4420197     {
4420198         if (endptr != NULL)
4420199             *endptr = (char *) &string[i];
4420200         return (number);
4420201     }
4420202 }
4420203 //
4420204 // The string is finished.
4420205 //
4420206 if (endptr != NULL)
4420207     *endptr = (char *) &string[i];
4420208 //
4420209 return (number);
4420210 }

```

## 95.19.20 lib/stdlib/strtoul.c

Si veda la sezione 88.130.

```

4430001 #include <stdlib.h>
4430002 #include <ctype.h>
4430003 #include <errno.h>
4430004 #include <limits.h>
4430005 //-----
4430006 // A really poor implementation. ,-(
4430007 //
4430008 unsigned long int
4430009 strtoul (const char *restrict string,
4430010         char **restrict endptr, int base)
4430011 {
4430012     return ((unsigned long int)
4430013           strtol (string, endptr, base));
4430014 }

```

## 95.19.21 lib/stdlib/unsetenv.c

Si veda la sezione 88.104.

```

4440001 #include <stdlib.h>
4440002 #include <string.h>
4440003 #include <errno.h>
4440004 //-----
4440005 extern char *_environment[];
4440006 extern char *_environment_table[];
4440007 //-----
4440008 int
4440009 unsetenv (const char *name)
4440010 {
4440011     int e;           // First index: environment table
4440012     // items.
4440013     int f;           // Second index: environment string
4440014     // scan.
4440015     //
4440016     // Check if the input is empty. No error is
4440017     // reported.
4440018     //
4440019     if (name == NULL || strlen (name) == 0)
4440020     {
4440021         return (0);
4440022     }
4440023     //
4440024     // Check if the input is valid: error here is
4440025     // reported.
4440026     //
4440027     if (strchr (name, '=') != NULL)
4440028     {
4440029         errset (EINVAL); // Invalid argument.
4440030         return (-1);
4440031     }
4440032     //
4440033     // Scan the environment table items, with index 'e'.
4440034     //
4440035     for (e = 0; e < ARG_MAX / 32; e++)
4440036     {
4440037         //
4440038         // Scan the string of the environment item, with
4440039         // index 'f'.
4440040         // The scan continue until 'name[f]' and
4440041         // '_environment[e][f]'
4440042         // are equal.
4440043         //
4440044         for (f = 0;

```



```

4440045     f < ARG_MAX / 16 - 1
4440046     && name[f] == _environment[e][f]; f++)
4440047     {
4440048         ; // Just scan.
4440049     }
4440050     //
4440051     // At this point, 'name[f]' and
4440052     // '_environment[e][f]' are
4440053     // different: if 'name[f]' is zero the name
4440054     // string is
4440055     // terminated; if '_environment[e][f]' is also
4440056     // equal to '=',
4440057     // the environment item is corresponding to the
4440058     // requested name.
4440059     //
4440060     if (name[f] == 0 && _environment[e][f] == '=')
4440061     {
4440062         //
4440063         // The environment item was found and it
4440064         // have to be removed.
4440065         // To be able to handle both 'setenv()' and
4440066         // 'putenv()',
4440067         // before removing the item, it is fixed the
4440068         // pointer to
4440069         // the global environment table.
4440070         //
4440071         _environment[e] = _environment_table[e];
4440072         //
4440073         // Now remove the environment item.
4440074         //
4440075         _environment[e][0] = 0;
4440076         break;
4440077     }
4440078 }
4440079 //
4440080 // Work done fine.
4440081 //
4440082 return (0);
4440083 }

```

## 95.19.22 lib/stdlib\_alloc/\_alloc\_list.c

Si veda la sezione 88.76.

```

4450001 #include <stdlib.h>
4450002 #include <stdio.h>
4450003 #include <unistd.h>
4450004 #include <stdint.h>
4450005 //-----
4450006 extern uintptr_t _alloc_start;
4450007 //-----
4450008 void
4450009 _alloc_list (void)
4450010 {
4450011     uintptr_t start = _alloc_start;
4450012     uintptr_t end = (uintptr_t) sbrk (0);
4450013     _alloc_head_t *head = (void *) start;
4450014     size_t actual_size;
4450015     uintptr_t current;
4450016     uintptr_t next;
4450017     uintptr_t up_to;
4450018     int counter;
4450019     //
4450020     // Scandisce la lista di blocchi di memoria.
4450021     //
4450022     counter = 2;
4450023     while (counter)
4450024     {
4450025         //
4450026         // Annota la posizione attuale e quella
4450027         // successiva.
4450028         //
4450029         current = (uintptr_t) head;
4450030         next = head->next * (sizeof (_alloc_head_t));
4450031         if (next == start)
4450032         {
4450033             up_to = end;
4450034         }
4450035         else
4450036         {
4450037             up_to = next;
4450038         }
4450039         //
4450040         // Se è stato raggiunto il primo elemento,
4450041         // decrementa il
4450042         // contatore di una unità. Se è già a zero,
4450043         // esce.

```

```

4450044 //
4450045     if (current == start)
4450046     {
4450047         counter--;
4450048         if (counter == 0)
4450049             break;
4450050     }
4450051     //
4450052     // Determina la dimensione del blocco attuale.
4450053     //
4450054     if (current == start && next == start)
4450055     {
4450056         //
4450057         // Si tratta del primo e unico elemento
4450058         // della lista.
4450059         //
4450060         actual_size =
4450061             end - start - (sizeof (_alloc_head_t));
4450062     }
4450063     else
4450064     {
4450065         actual_size =
4450066             up_to - current - (sizeof (_alloc_head_t));
4450067     }
4450068     //
4450069     // Si mostra lo stato del blocco di memoria.
4450070     //
4450071     if (head->allocated)
4450072     {
4450073         printf ("[%s] used %08X..%08X size %08zX\n",
4450074             __func__,
4450075             current + (sizeof (_alloc_head_t)),
4450076             up_to, actual_size);
4450077     }
4450078     else
4450079     {
4450080         printf ("[%s] free %08X..%08X size %08zX\n",
4450081             __func__,
4450082             current + (sizeof (_alloc_head_t)),
4450083             up_to, actual_size);
4450084     }
4450085     //
4450086     // Si passa alla posizione successiva.
4450087     //
4450088     head = (void *) next;
4450089 }
4450090 }

```

## 95.19.23 lib/stdlib\_alloc/free.c

Si veda la sezione 88.76.

```

4460001 #include <stdlib.h>
4460002 #include <stdio.h>
4460003 #include <unistd.h>
4460004 //-----
4460005 extern uintptr_t _alloc_start;
4460006 //-----
4460007 void
4460008 free (void *ptr)
4460009 {
4460010     _alloc_head_t *start = (_alloc_head_t *) _alloc_start;
4460011     _alloc_head_t *head_current = ((_alloc_head_t *) ptr) - 1;
4460012     _alloc_head_t *head_next;
4460013     //
4460014     // Verifica il blocco attuale e, se è possibile, lo
4460015     // libera.
4460016     //
4460017     if (head_current->allocated == 1)
4460018     {
4460019         head_current->allocated = 0;
4460020     }
4460021     else
4460022     {
4460023         printf ("[%s] ERROR: cannot free %08X!\n",
4460024             __func__,
4460025             (uintptr_t) head_current +
4460026             (sizeof (_alloc_head_t)));
4460027     }
4460028     //
4460029     // Scandisce i blocchi liberi, cercando quelli
4460030     // adiacenti per
4460031     // allungarli. Se il blocco successivo è il primo,
4460032     // termina,
4460033     // perché non può avvenire alcuna fusione con
4460034     // quello precedente.
4460035     //

```

```

446036 head_current = start;
446037 while (1)
446038 {
446039     //
446040     // Individua il blocco successivo.
446041     //
446042     head_next =
446043     (_alloc_head_t *) (head_current->next
446044     * (sizeof (_alloc_head_t)));
446045     //
446046     // Controlla se è il primo.
446047     //
446048     if (head_next == start)
446049     {
446050         break;
446051     }
446052     //
446053     //
446054     //
446055     if (head_current->allocated == 0)
446056     {
446057         //
446058         // Controlla se si può espandere.
446059         //
446060         if (head_next->allocated == 0)
446061         {
446062             head_current->next = head_next->next;
446063         }
446064         else
446065         {
446066             head_current = head_next;
446067         }
446068     }
446069     else
446070     {
446071         head_current = (_alloc_head_t *)
446072         (head_current->next * (sizeof (_alloc_head_t)));
446073     }
446074 }
446075 }

```

#### 95.19.24 lib/stdlib\_alloc/malloc.c

«

Si veda la sezione 88.76.

```

447001 #include <stdlib.h>
447002 #include <unistd.h>
447003 #include <errno.h>
447004 //-----
447005 uintptr_t _alloc_start = 0;
447006 //-----
447007 static int _alloc_init (void);
447008 static void *_malloc (size_t size);
447009 //-----
447010 void *
447011 malloc (size_t size)
447012 {
447013     void *pstatus;
447014     int status;
447015     //
447016     // Verify to have initialized the allocation memory.
447017     //
447018     if (_alloc_start == 0)
447019     {
447020         status = _alloc_init ();
447021         if (status < 0)
447022         {
447023             errset (ENOMEM);
447024             return (NULL);
447025         }
447026     }
447027     //
447028     // Try to allocate as usual.
447029     //
447030     pstatus = _malloc (size);
447031     //
447032     if (pstatus == NULL)
447033     {
447034         //
447035         // Try to increase memory for the process.
447036         //
447037         pstatus = sbrk (size);
447038         if (pstatus == NULL)
447039         {
447040             //
447041             // Sorry: no way to get memory.
447042             //

```

```

447043         errset (ENOMEM);
447044         return (NULL);
447045     }
447046     //
447047     // Ok. Now try again to allocate memory.
447048     //
447049     return (_malloc (size));
447050 }
447051 else
447052 {
447053     //
447054     // The first allocation was successful.
447055     //
447056     return (pstatus);
447057 }
447058 }
447059 //-----
447060 static int
447061 _alloc_init (void)
447062 {
447063     {
447064         uintptr_t start;
447065         uintptr_t end;
447066         _alloc_head_t *head;
447067         size_t available;
447068         //
447069         // Get size.
447070         //
447071         if (_alloc_start == 0)
447072         {
447073             _alloc_start = (uintptr_t) sbrk (0);
447074         }
447075         //
447076         start = _alloc_start;
447077         end = (uintptr_t) sbrk (0);
447078         available = end - start;
447079         //
447080         // Check available space.
447081         //
447082         if (available < ((sizeof (_alloc_head_t)) * 2))
447083         {
447084             //
447085             // Try to get a little memory.
447086             //
447087             sbrk ((sizeof (_alloc_head_t)) * 2);
447088             end = (uintptr_t) sbrk (0);
447089             available = end - start;
447090             if (available < ((sizeof (_alloc_head_t)) * 2))
447091             {
447092                 //
447093                 // Sorry!
447094                 //
447095                 return (-1);
447096             }
447097         }
447098         //
447099         // Prepare the list main node.
447100         //
447101         head = (_alloc_head_t *) start;
447102         //
447103         // Init the first free block, that points to itself,
447104         // as it is
447105         // the only one.
447106         //
447107         head->allocated = 0;
447108         head->next = (start / (sizeof (_alloc_head_t)));
447109         //
447110         // Ok.
447111         //
447112         return (0);
447113     }
447114 }
447115 //-----
447116 static void *
447117 _malloc (size_t size)
447118 {
447119     uintptr_t start = _alloc_start;
447120     uintptr_t end = (uintptr_t) sbrk (0);
447121     _alloc_head_t *head = (void *) start;
447122     size_t actual_size;
447123     uintptr_t current;
447124     uintptr_t next;
447125     uintptr_t new;
447126     uintptr_t up_to;
447127     int counter;
447128     //
447129     // Arrotonda in eccesso il valore di «size», in

```

```

4470130 // modo che sia un
4470131 // multiplo della dimensione di «_alloc_head_t».
4470132 // Altrimenti, la
4470133 // collocazione dei blocchi successivi può avvenire
4470134 // in modo
4470135 // non allineato.
4470136 //
4470137 size = (size + (sizeof (_alloc_head_t) - 1);
4470138 size = size / (sizeof (_alloc_head_t));
4470139 size = size * (sizeof (_alloc_head_t));
4470140 //
4470141 // Cerca un blocco libero di dimensione sufficiente.
4470142 //
4470143 counter = 2;
4470144 while (counter)
4470145 {
4470146 //
4470147 // Annota la posizione attuale e quella
4470148 // successiva.
4470149 //
4470150 current = (uintptr_t) head;
4470151 next = head->next * (sizeof (_alloc_head_t));
4470152 //
4470153 if (next == start)
4470154 {
4470155     up_to = end;
4470156 }
4470157 else
4470158 {
4470159     up_to = next;
4470160 }
4470161 //
4470162 // Se è stato raggiunto il primo elemento,
4470163 // decrementa il
4470164 // contatore di una unità. Se è già a zero,
4470165 // esce.
4470166 //
4470167 if (current == start)
4470168 {
4470169     counter--;
4470170     if (counter == 0)
4470171         break;
4470172 }
4470173 //
4470174 // Controlla se si tratta di un blocco libero.
4470175 //
4470176
4470177 if (!head->allocated)
4470178 {
4470179 //
4470180 // Il blocco è libero: si deve determinarne
4470181 // la dimensione.
4470182 //
4470183 if (current == start && next == start)
4470184 {
4470185 //
4470186 // Si tratta del primo e unico elemento
4470187 // della lista.
4470188 //
4470189 actual_size =
4470190     end - start - (sizeof (_alloc_head_t));
4470191 }
4470192 else
4470193 {
4470194     actual_size =
4470195         up_to - current - (sizeof (_alloc_head_t));
4470196 }
4470197 //
4470198 // Si verifica che sia capiente.
4470199 //
4470200 if (actual_size >=
4470201     size + ((sizeof (_alloc_head_t) * 2))
4470202 {
4470203 //
4470204 // C'è spazio per dividere il blocco.
4470205 //
4470206 new =
4470207     current + size + (sizeof (_alloc_head_t));
4470208 //
4470209 // Aggiorna l'intestazione attuale.
4470210 //
4470211 head->allocated = 1;
4470212 head->next = new / (sizeof (_alloc_head_t));
4470213 //
4470214 // Predispone l'intestazione successiva.
4470215 //
4470216 head = (void *) new;

```

```

4470217     head->allocated = 0;
4470218     head->next = next / (sizeof (_alloc_head_t));
4470219 //
4470220 // Restituisce l'indirizzo iniziale
4470221 // dello spazio libero,
4470222 // successivo all'intestazione.
4470223 //
4470224     return (void *) (current +
4470225         (sizeof (_alloc_head_t)));
4470226 }
4470227 else if (actual_size >= size)
4470228 {
4470229 //
4470230 // Il blocco va usato per intero.
4470231 //
4470232     head->allocated = 1;
4470233 //
4470234 // Restituisce l'indirizzo iniziale
4470235 // dello spazio libero,
4470236 // successivo all'intestazione.
4470237 //
4470238     return (void *) (current +
4470239         (sizeof (_alloc_head_t)));
4470240 }
4470241 }
4470242 //
4470243 // Il blocco è allocato, oppure è di
4470244 // dimensione insufficiente;
4470245 // pertanto occorre passare alla posizione
4470246 // successiva.
4470247 //
4470248     head = (void *) next;
4470249 }
4470250 //
4470251 // Essendo terminato il ciclo precedente, vuol dire
4470252 // che non ci sono spazi disponibili.
4470253 //
4470254     errset (ENOMEM);
4470255     return NULL;
4470256 }

```

## 95.19.25 lib/stdlib\_alloc/realloc.c

Si veda la sezione 88.76.

```

4480001 #include <stdlib.h>
4480002 #include <stdio.h>
4480003 #include <unistd.h>
4480004 #include <string.h>
4480005 //-----
4480006 extern uintptr_t _alloc_start;
4480007 //-----
4480008 void *
4480009 realloc (void *ptr, size_t size)
4480010 {
4480011     uintptr_t start = _alloc_start;
4480012     uintptr_t end = (uintptr_t) sbrk (0);
4480013     size_t actual_size;
4480014     _alloc_head_t *head = ((_alloc_head_t *) ptr) - 1;
4480015     _alloc_head_t *head_new;
4480016     void *ptr_new;
4480017 //
4480018 // Verifica che il puntatore riguardi effettivamente
4480019 // un'area occupata.
4480020 //
4480021 if (!head->allocated)
4480022 {
4480023     printf
4480024         ("%s] ERROR: cannot re-allocate %08X that is "
4480025         "not already allocated!", __func__,
4480026         (uintptr_t) ptr);
4480027 }
4480028 //
4480029 // Arrotonda in eccesso il valore di «size», in
4480030 // modo che sia un
4480031 // multiplo della dimensione di «_alloc_head_t».
4480032 // Altrimenti, la
4480033 // collocazione dei blocchi successivi può avvenire
4480034 // in modo
4480035 // non allineato.
4480036 //
4480037 size = (size + (sizeof (_alloc_head_t) - 1);
4480038 size = size / (sizeof (_alloc_head_t));
4480039 size = size * (sizeof (_alloc_head_t));
4480040 //
4480041 // Determina la dimensione attuale.
4480042 //

```

```

4480043 if ((head->next * (sizeof (_alloc_head_t))) == start)
4480044 {
4480045     actual_size = end - ((uintptr_t) ptr);
4480046 }
4480047 else
4480048 {
4480049     actual_size =
4480050     (head->next * (sizeof (_alloc_head_t))) -
4480051     ((uintptr_t) ptr);
4480052 }
4480053 //
4480054 // Se la dimensione richiesta è inferiore, può
4480055 // ridurre
4480056 // l'estensione del blocco.
4480057 //
4480058 if (size == actual_size)
4480059 {
4480060     return ptr;
4480061 }
4480062 else if (size <=
4480063     (actual_size - (sizeof (_alloc_head_t)) * 2))
4480064 {
4480065     //
4480066     // Si può ricavare lo spazio libero rimanente.
4480067     //
4480068     head_new = (_alloc_head_t *) (((char *) ptr) + size);
4480069     //
4480070     head_new->next = head->next;
4480071     head_new->allocated = 0;
4480072     //
4480073     head->next =
4480074     ((uintptr_t) head_new) / (sizeof (_alloc_head_t));
4480075     //
4480076     return ptr;
4480077 }
4480078 else if (size < actual_size)
4480079 {
4480080     //
4480081     // Anche se è minore, non si può ridurre lo
4480082     // spazio usato
4480083     // effettivamente.
4480084     //
4480085     return ptr;
4480086 }
4480087 else
4480088 {
4480089     //
4480090     // La dimensione richiesta è maggiore.
4480091     //
4480092     ptr_new = malloc (size);
4480093     //
4480094     if (ptr_new)
4480095     {
4480096         //
4480097         // Ricopia i dati nella nuova collocazione.
4480098         //
4480099         memcpy (ptr_new, ptr, actual_size);
4480100         //
4480101         // Libera la collocazione vecchia.
4480102         //
4480103         free (ptr);
4480104         //
4480105         return ptr_new;
4480106     }
4480107     else
4480108     {
4480109         return NULL;
4480110     }
4480111 }
4480112 }

```

## 95.20 os32: «lib/string.h»

«

Si veda la sezione 91.3.

```

4490001 #ifndef _STRING_H
4490002 #define _STRING_H    1
4490003 //-----
4490004 #include <size_t.h>
4490005 #include <NULL.h>
4490006 #include <restrict.h>
4490007 //-----
4490008 void *memcpy (void *restrict dst,
4490009     const void *restrict org, int c, size_t n);
4490010 void *memchr (const void *memory, int c, size_t n);
4490011 int memcmp (const void *memory1, const void *memory2,
4490012     size_t n);

```

```

4490013 void *memcpy (void *restrict dst,
4490014     const void *restrict org, size_t n);
4490015 void *memmove (void *dst, const void *org, size_t n);
4490016 void *memset (void *memory, int c, size_t n);
4490017 char *strcat (char *restrict dst, const char *restrict org);
4490018 char *strchr (const char *string, int c);
4490019 int strcmp (const char *string1, const char *string2);
4490020 int strcoll (const char *string1, const char *string2);
4490021 char *strcpy (char *restrict dst, const char *restrict org);
4490022 size_t strcspn (const char *string, const char *reject);
4490023 char *strdup (const char *string);
4490024 char *strerror (int errnum);
4490025 size_t strlen (const char *string);
4490026 char *strncat (char *restrict dst,
4490027     const char *restrict org, size_t n);
4490028 int strncmp (const char *string1, const char *string2,
4490029     size_t n);
4490030 char *strncpy (char *restrict dst,
4490031     const char *restrict org, size_t n);
4490032 char *strpbrk (const char *string, const char *accept);
4490033 char *strrchr (const char *string, int c);
4490034 size_t strspn (const char *string, const char *accept);
4490035 char *strstr (const char *string, const char *substring);
4490036 char *strtok (char *restrict string,
4490037     const char *restrict delim);
4490038 size_t strxfrm (char *restrict dst,
4490039     const char *restrict org, size_t n);
4490040 //-----
4490041
4490042 #endif

```

95.20.1	lib/string/memccpy.c	867
95.20.2	lib/string/memchr.c	868
95.20.3	lib/string/memcmp.c	868
95.20.4	lib/string/memcpy.c	868
95.20.5	lib/string/memmove.c	868
95.20.6	lib/string/memset.c	869
95.20.7	lib/string/strcat.c	869
95.20.8	lib/string/strchr.c	869
95.20.9	lib/string/strcmp.c	870
95.20.10	lib/string/strcoll.c	870
95.20.11	lib/string/strcpy.c	870
95.20.12	lib/string/strcspn.c	870
95.20.13	lib/string/strdup.c	871
95.20.14	lib/string/strerror.c	871
95.20.15	lib/string/strlen.c	873
95.20.16	lib/string/strncat.c	873
95.20.17	lib/string/strncmp.c	873
95.20.18	lib/string/strncpy.c	873
95.20.19	lib/string/strpbrk.c	874
95.20.20	lib/string/strrchr.c	874
95.20.21	lib/string/strspn.c	874
95.20.22	lib/string/strstr.c	875
95.20.23	lib/string/strtok.c	875
95.20.24	lib/string/strxfrm.c	877

## 95.20.1 lib/string/memccpy.c

Si veda la sezione 88.77.

»

```

4500001 #include <string.h>
4500002 //-----
4500003 void *
4500004 memcpy (void *restrict dst, const void *restrict org,
4500005     int c, size_t n)
4500006 {
4500007     char *d = (char *) dst;
4500008     char *o = (char *) org;
4500009     size_t i;
4500010     for (i = 0; n > 0 && i < n; i++)

```

```

450011 {
450012     d[i] = o[i];
450013     if (d[i] == (char) c)
450014     {
450015         return ((void *) &d[i + 1]);
450016     }
450017 }
450018 return (NULL);
450019 }

```

### 95.20.2 lib/string/memchr.c

&lt;&lt;

Si veda la sezione 88.78.

```

451001 #include <string.h>
451002 //-----
451003 void *
451004 memchr (const void *memory, int c, size_t n)
451005 {
451006     char *m = (char *) memory;
451007     size_t i;
451008     for (i = 0; n > 0 && i < n; i++)
451009     {
451010         if (m[i] == (char) c)
451011         {
451012             return (void *) (m + i);
451013         }
451014     }
451015     return NULL;
451016 }

```

### 95.20.3 lib/string/memcmp.c

&lt;&lt;

Si veda la sezione 88.79.

```

452001 #include <string.h>
452002 //-----
452003 int
452004 memcmp (const void *memory1, const void *memory2, size_t n)
452005 {
452006     char *a = (char *) memory1;
452007     char *b = (char *) memory2;
452008     size_t i;
452009     for (i = 0; n > 0 && i < n; i++)
452010     {
452011         if (a[i] > b[i])
452012         {
452013             return 1;
452014         }
452015         else if (a[i] < b[i])
452016         {
452017             return -1;
452018         }
452019     }
452020     return 0;
452021 }

```

### 95.20.4 lib/string/memcpy.c

&lt;&lt;

Si veda la sezione 88.80.

```

453001 #include <string.h>
453002 //-----
453003 void *
453004 memcpy (void *restrict dst, const void *restrict org,
453005         size_t n)
453006 {
453007     char *d = (char *) dst;
453008     char *o = (char *) org;
453009     size_t i;
453010     for (i = 0; n > 0 && i < n; i++)
453011     {
453012         d[i] = o[i];
453013     }
453014     return dst;
453015 }

```

### 95.20.5 lib/string/memmove.c

&lt;&lt;

Si veda la sezione 88.81.

```

454001 #include <string.h>
454002 //-----
454003 void *
454004 memmove (void *dst, const void *org, size_t n)
454005 {

```

```

454006     char *d = (char *) dst;
454007     char *o = (char *) org;
454008     size_t i;
454009     //
454010     // Depending on the memory start locations, copy may
454011     // be direct or
454012     // reverse, to avoid overwriting before the
454013     // relocation is done.
454014     //
454015     if (d < o)
454016     {
454017         for (i = 0; i < n; i++)
454018         {
454019             d[i] = o[i];
454020         }
454021     }
454022     else if (d == o)
454023     {
454024         //
454025         // Memory locations are already the same.
454026         //
454027         ;
454028     }
454029     else
454030     {
454031         for (i = n - 1; i >= 0; i--)
454032         {
454033             d[i] = o[i];
454034         }
454035     }
454036     return dst;
454037 }

```

### 95.20.6 lib/string/memset.c

&lt;&lt;

Si veda la sezione 88.82.

```

455001 #include <string.h>
455002 //-----
455003 void *
455004 memset (void *memory, int c, size_t n)
455005 {
455006     char *m = (char *) memory;
455007     size_t i;
455008     for (i = 0; n > 0 && i < n; i++)
455009     {
455010         m[i] = (char) c;
455011     }
455012     return memory;
455013 }

```

### 95.20.7 lib/string/strcat.c

&lt;&lt;

Si veda la sezione 88.113.

```

456001 #include <string.h>
456002 //-----
456003 char *
456004 strcat (char *restrict dst, const char *restrict org)
456005 {
456006     size_t i;
456007     size_t j;
456008     for (i = 0; dst[i] != 0; i++)
456009     {
456010         ; // Just look for the null character.
456011     }
456012     for (j = 0; org[j] != 0; j++)
456013     {
456014         dst[i] = org[j];
456015     }
456016     dst[i] = 0;
456017     return dst;
456018 }

```

### 95.20.8 lib/string/strchr.c

&lt;&lt;

Si veda la sezione 88.114.

```

457001 #include <string.h>
457002 //-----
457003 char *
457004 strchr (const char *string, int c)
457005 {
457006     size_t i;
457007     for (i = 0; i++)
457008     {
457009         if (string[i] == (char) c)

```

```

4570010     {
4570011         return (char *) (string + i);
4570012     }
4570013     else if (string[i] == 0)
4570014     {
4570015         return NULL;
4570016     }
4570017 }
4570018 }

```

## 95.20.9 lib/string/stricmp.c

« Si veda la sezione 88.115.

```

4580001 #include <string.h>
4580002 //-----
4580003 int
4580004 strcmp (const char *string1, const char *string2)
4580005 {
4580006     char *a = (char *) string1;
4580007     char *b = (char *) string2;
4580008     size_t i;
4580009     for (i = 0;; i++)
4580010     {
4580011         if (a[i] > b[i])
4580012         {
4580013             return 1;
4580014         }
4580015         else if (a[i] < b[i])
4580016         {
4580017             return -1;
4580018         }
4580019         else if (a[i] == 0 && b[i] == 0)
4580020         {
4580021             return 0;
4580022         }
4580023     }
4580024 }

```

## 95.20.10 lib/string/strcoll.c

« Si veda la sezione 88.115.

```

4590001 #include <string.h>
4590002 //-----
4590003 int
4590004 strcoll (const char *string1, const char *string2)
4590005 {
4590006     return (strcmp (string1, string2));
4590007 }

```

## 95.20.11 lib/string/strcpy.c

« Si veda la sezione 88.117.

```

4600001 #include <string.h>
4600002 //-----
4600003 char *
4600004 strcpy (char *restrict dst, const char *restrict org)
4600005 {
4600006     size_t i;
4600007     for (i = 0; org[i] != 0; i++)
4600008     {
4600009         dst[i] = org[i];
4600010     }
4600011     dst[i] = 0;
4600012     return dst;
4600013 }

```

## 95.20.12 lib/string/strcspn.c

« Si veda la sezione 88.127.

```

4610001 #include <string.h>
4610002 //-----
4610003 size_t
4610004 strcspn (const char *string, const char *reject)
4610005 {
4610006     size_t i;
4610007     size_t j;
4610008     int found;
4610009     for (i = 0; string[i] != 0; i++)
4610010     {
4610011         for (j = 0, found = 0; reject[j] != 0 || found; j++)
4610012         {
4610013             if (string[i] == reject[j])

```

```

4610014         {
4610015             found = 1;
4610016             break;
4610017         }
4610018     }
4610019     if (found)
4610020     {
4610021         break;
4610022     }
4610023 }
4610024 return i;
4610025 }

```

## 95.20.13 lib/string/strdup.c

« Si veda la sezione 88.119.

```

4620001 #include <string.h>
4620002 #include <stdlib.h>
4620003 #include <errno.h>
4620004 //-----
4620005 char *
4620006 strdup (const char *string)
4620007 {
4620008     size_t size;
4620009     char *copy;
4620010     //
4620011     // Get string size: must be added 1, to count the
4620012     // termination null
4620013     // character.
4620014     //
4620015     size = strlen (string) + 1;
4620016     //
4620017     copy = malloc (size);
4620018     //
4620019     if (copy == NULL)
4620020     {
4620021         errset (ENOMEM); // Not enough memory.
4620022         return (NULL);
4620023     }
4620024     //
4620025     strcpy (copy, string);
4620026     //
4620027     return (copy);
4620028 }

```

## 95.20.14 lib/string/strerror.c

« Si veda la sezione 88.120.

```

4630001 #include <string.h>
4630002 #include <errno.h>
4630003 //-----
4630004 #define ERROR_MAX 120
4630005 //-----
4630006 char *
4630007 strerror (int errnum)
4630008 {
4630009     static char *err[ERROR_MAX];
4630010     //
4630011     err[0] = "No error";
4630012     err[E2BIG] = TEXT_E2BIG;
4630013     err[EACCES] = TEXT_EACCES;
4630014     err[EADDRINUSE] = TEXT_EADDRINUSE;
4630015     err[EADDRNOTAVAIL] = TEXT_EADDRNOTAVAIL;
4630016     err[EAFNOSUPPORT] = TEXT_EAFNOSUPPORT;
4630017     err[EAGAIN] = TEXT_EAGAIN;
4630018     err[EALREADY] = TEXT_EALREADY;
4630019     err[EBADF] = TEXT_EBADF;
4630020     err[EBADMSG] = TEXT_EBADMSG;
4630021     err[EBUSY] = TEXT_EBUSY;
4630022     err[ECANCELED] = TEXT_ECANCELED;
4630023     err[ECHILD] = TEXT_ECHILD;
4630024     err[ECONNABORTED] = TEXT_ECONNABORTED;
4630025     err[ECONNREFUSED] = TEXT_ECONNREFUSED;
4630026     err[ECONNRESET] = TEXT_ECONNRESET;
4630027     err[EDEADLK] = TEXT_EDEADLK;
4630028     err[EDESTADDRREQ] = TEXT_EDESTADDRREQ;
4630029     err[EDOM] = TEXT_EDOM;
4630030     err[EDQUOT] = TEXT_EDQUOT;
4630031     err[EEXIST] = TEXT_EEXIST;
4630032     err[EFAULT] = TEXT_EFAULT;
4630033     err[EFBIG] = TEXT_EFBIG;
4630034     err[EHOSTUNREACH] = TEXT_EHOSTUNREACH;
4630035     err[EIDRM] = TEXT_EIDRM;
4630036     err[EILSEQ] = TEXT_EILSEQ;
4630037     err[EINPROGRESS] = TEXT_EINPROGRESS;

```

```

4630038 err[EINTR] = TEXT_EINTR;
4630039 err[EINVAL] = TEXT_EINVAL;
4630040 err[EIO] = TEXT_EIO;
4630041 err[EISCONN] = TEXT_EISCONN;
4630042 err[EISDIR] = TEXT_EISDIR;
4630043 err[ELOOP] = TEXT_ELOOP;
4630044 err[EMFILE] = TEXT_EMFILE;
4630045 err[EMLINK] = TEXT_EMLINK;
4630046 err[EMSGSIZE] = TEXT_EMSGSIZE;
4630047 err[EMULTIHOP] = TEXT_EMULTIHOP;
4630048 err[ENAMETOOLONG] = TEXT_ENAMETOOLONG;
4630049 err[ENETDOWN] = TEXT_ENETDOWN;
4630050 err[ENETRESET] = TEXT_ENETRESET;
4630051 err[ENETUNREACH] = TEXT_ENETUNREACH;
4630052 err[ENFILE] = TEXT_ENFILE;
4630053 err[ENOBUFS] = TEXT_ENOBUFS;
4630054 err[ENODATA] = TEXT_ENODATA;
4630055 err[ENODEV] = TEXT_ENODEV;
4630056 err[ENOENT] = TEXT_ENOENT;
4630057 err[ENOEXEC] = TEXT_ENOEXEC;
4630058 err[ENOLCK] = TEXT_ENOLCK;
4630059 err[ENOLINK] = TEXT_ENOLINK;
4630060 err[ENOMEM] = TEXT_ENOMEM;
4630061 err[ENOMSG] = TEXT_ENOMSG;
4630062 err[ENOPROTOOPT] = TEXT_ENOPROTOOPT;
4630063 err[ENOSPC] = TEXT_ENOSPC;
4630064 err[ENOSR] = TEXT_ENOSR;
4630065 err[ENOSTR] = TEXT_ENOSTR;
4630066 err[ENOSYS] = TEXT_ENOSYS;
4630067 err[ENOTCONN] = TEXT_ENOTCONN;
4630068 err[ENOTDIR] = TEXT_ENOTDIR;
4630069 err[ENOTEMPTY] = TEXT_ENOTEMPTY;
4630070 err[ENOTSOCK] = TEXT_ENOTSOCK;
4630071 err[ENOTSUP] = TEXT_ENOTSUP;
4630072 err[ENOTTY] = TEXT_ENOTTY;
4630073 err[ENXIO] = TEXT_ENXIO;
4630074 err[EOPNOTSUPP] = TEXT_EOPNOTSUPP;
4630075 err[E_OVERFLOW] = TEXT_E_OVERFLOW;
4630076 err[EPERM] = TEXT_EPERM;
4630077 err[EPIPE] = TEXT_EPIPE;
4630078 err[EPROTO] = TEXT_EPROTO;
4630079 err[EPROTONOSUPPORT] = TEXT_EPROTONOSUPPORT;
4630080 err[EPROTOTYPE] = TEXT_EPROTOTYPE;
4630081 err[ERANGE] = TEXT_ERANGE;
4630082 err[EROFS] = TEXT_EROFS;
4630083 err[ESPIPE] = TEXT_ESPIPE;
4630084 err[ESRCH] = TEXT_ESRCH;
4630085 err[ESTALE] = TEXT_ESTALE;
4630086 err[ETIME] = TEXT_ETIME;
4630087 err[ETIMEDOUT] = TEXT_ETIMEDOUT;
4630088 err[ETXTBSY] = TEXT_ETXTBSY;
4630089 err[EWOULDBLOCK] = TEXT_EWOULDBLOCK;
4630090 err[EXDEV] = TEXT_EXDEV;
4630091 err[E_NO_MEDIUM] = TEXT_E_NO_MEDIUM;
4630092 err[E_MEDIUM] = TEXT_E_MEDIUM;
4630093 err[E_FILE_TYPE] = TEXT_E_FILE_TYPE;
4630094 err[E_ROOT_INODE_NOT_CACHED] =
    TEXT_E_ROOT_INODE_NOT_CACHED;
4630095 err[E_CANNOT_READ_SUPERBLOCK] =
    TEXT_E_CANNOT_READ_SUPERBLOCK;
4630096 err[E_MAP_INODE_TOO_BIG] = TEXT_E_MAP_INODE_TOO_BIG;
4630097 err[E_MAP_ZONE_TOO_BIG] = TEXT_E_MAP_ZONE_TOO_BIG;
4630098 err[E_DATA_ZONE_TOO_BIG] = TEXT_E_DATA_ZONE_TOO_BIG;
4630099 err[E_CANNOT_FIND_ROOT_DEVICE] =
    TEXT_E_CANNOT_FIND_ROOT_DEVICE;
4630100 err[E_CANNOT_FIND_ROOT_INODE] =
    TEXT_E_CANNOT_FIND_ROOT_INODE;
4630101 err[E_FILE_TYPE_UNSUPPORTED] =
    TEXT_E_FILE_TYPE_UNSUPPORTED;
4630102 err[E_ENV_TOO_BIG] = TEXT_E_ENV_TOO_BIG;
4630103 err[E_LIMIT] = TEXT_E_LIMIT;
4630104 err[E_NOT_MOUNTED] = TEXT_E_NOT_MOUNTED;
4630105 err[E_NOT_IMPLEMENTED] = TEXT_E_NOT_IMPLEMENTED;
4630106 err[E_HARDWARE_FAULT] = TEXT_E_HARDWARE_FAULT;
4630107 err[E_DRIVER_FAULT] = TEXT_E_DRIVER_FAULT;
4630108 err[E_PIPE_FULL] = TEXT_E_PIPE_FULL;
4630109 err[E_PIPE_EMPTY] = TEXT_E_PIPE_EMPTY;
4630110 err[E_PART_TYPE_NOT_MINIX] = TEXT_E_PART_TYPE_NOT_MINIX;
4630111 err[E_FS_TYPE_NOT_SUPPORTED] =
    TEXT_E_FS_TYPE_NOT_SUPPORTED;
4630112 err[E_PDU_TOO_BIG] = TEXT_E_PDU_TOO_BIG;
4630113 err[E_ARP_MISSING] = TEXT_E_ARP_MISSING;
4630114 //
4630115 if (errno >= ERROR_MAX || errno < 0)
4630116 {
4630117     return ("Unknown error");
4630118 }
4630119

```

```

4630125 //
4630126     return (err[errno]);
4630127 }

```

## 95.20.15 lib/string/strlen.c

Si veda la sezione 88.121. «

```

4640001 #include <string.h>
4640002 //-----
4640003 size_t
4640004 strlen (const char *string)
4640005 {
4640006     size_t i;
4640007     for (i = 0; string[i] != 0; i++)
4640008     {
4640009         ; // Just count.
4640010     }
4640011     return i;
4640012 }

```

## 95.20.16 lib/string/strncat.c

Si veda la sezione 88.113. «

```

4650001 #include <string.h>
4650002 //-----
4650003 char *
4650004 strncat (char *restrict dst, const char *restrict org,
4650005         size_t n)
4650006 {
4650007     size_t i;
4650008     size_t j;
4650009     for (i = 0; n > 0 && dst[i] != 0; i++)
4650010     {
4650011         ; // Just seek the null character.
4650012     }
4650013     for (j = 0; n > 0 && j < n && org[j] != 0; j++)
4650014     {
4650015         dst[i] = org[j];
4650016     }
4650017     dst[i] = 0;
4650018     return dst;
4650019 }

```

## 95.20.17 lib/string/strncmp.c

Si veda la sezione 88.115. «

```

4660001 #include <string.h>
4660002 //-----
4660003 int
4660004 strncmp (const char *string1, const char *string2, size_t n)
4660005 {
4660006     size_t i;
4660007     for (i = 0; i < n; i++)
4660008     {
4660009         if (string1[i] > string2[i])
4660010         {
4660011             return 1;
4660012         }
4660013         else if (string1[i] < string2[i])
4660014         {
4660015             return -1;
4660016         }
4660017         else if (string1[i] == 0 && string2[i] == 0)
4660018         {
4660019             return 0;
4660020         }
4660021     }
4660022     return 0;
4660023 }

```

## 95.20.18 lib/string/strncpy.c

Si veda la sezione 88.117. «

```

4670001 #include <string.h>
4670002 //-----
4670003 char *
4670004 strncpy (char *restrict dst, const char *restrict org,
4670005         size_t n)
4670006 {
4670007     size_t i;
4670008     for (i = 0; n > 0 && i < n && org[i] != 0; i++)
4670009     {

```

```

4670010     dst[i] = org[i];
4670011     }
4670012     for (; n > 0 && i < n; i++)
4670013     {
4670014         dst[i] = 0;
4670015     }
4670016     return dst;
4670017 }

```

## 95.20.19 lib/string/strpbrk.c

«

Si veda la sezione 88.125.

```

4680001 #include <string.h>
4680002 //-----
4680003 char *
4680004 strpbrk (const char *string, const char *accept)
4680005 {
4680006     //
4680007     // The first parameter not 'const char *' because
4680008     // otherwise
4680009     // the return value should be 'const char *' too!
4680010     //
4680011     size_t i;
4680012     size_t j;
4680013     //
4680014     for (i = 0; string[i] != 0; i++)
4680015     {
4680016         for (j = 0; accept[j] != 0; j++)
4680017         {
4680018             if (string[i] == accept[j])
4680019             {
4680020                 return (char *) (string + i);
4680021             }
4680022         }
4680023     }
4680024     return NULL;
4680025 }

```

## 95.20.20 lib/string/strchr.c

«

Si veda la sezione 88.114.

```

4690001 #include <string.h>
4690002 //-----
4690003 char *
4690004 strchr (const char *string, int c)
4690005 {
4690006     int i;
4690007     for (i = strlen (string); i >= 0; i--)
4690008     {
4690009         if (string[i] == (char) c)
4690010         {
4690011             break;
4690012         }
4690013     }
4690014     if (i < 0)
4690015     {
4690016         return NULL;
4690017     }
4690018     else
4690019     {
4690020         return (char *) (string + i);
4690021     }
4690022 }

```

## 95.20.21 lib/string/strspn.c

«

Si veda la sezione 88.127.

```

4700001 #include <string.h>
4700002 //-----
4700003 size_t
4700004 strspn (const char *string, const char *accept)
4700005 {
4700006     size_t i;
4700007     size_t j;
4700008     int found;
4700009     for (i = 0; string[i] != 0; i++)
4700010     {
4700011         for (j = 0, found = 0; accept[j] != 0; j++)
4700012         {
4700013             if (string[i] == accept[j])
4700014             {
4700015                 found = 1;
4700016                 break;
4700017             }

```

```

4700018     }
4700019     if (!found)
4700020     {
4700021         break;
4700022     }
4700023     }
4700024     return i;
4700025 }

```

## 95.20.22 lib/string/strstr.c

»

Si veda la sezione 88.128.

```

4710001 #include <string.h>
4710002 //-----
4710003 char *
4710004 strstr (const char *string, const char *substring)
4710005 {
4710006     size_t i;
4710007     size_t j;
4710008     size_t k;
4710009     int found;
4710010     if (substring[0] == 0)
4710011     {
4710012         return (char *) string;
4710013     }
4710014     for (i = 0, j = 0, found = 0; string[i] != 0; i++)
4710015     {
4710016         if (string[i] == substring[0])
4710017         {
4710018             for (k = i, j = 0;
4710019                  string[k] == substring[j] &&
4710020                  string[k] != 0 &&
4710021                  substring[j] != 0; j++, k++)
4710022             {
4710023                 ;
4710024             }
4710025             if (substring[j] == 0)
4710026             {
4710027                 found = 1;
4710028             }
4710029         }
4710030     }
4710031     if (found)
4710032     {
4710033         return (char *) (string + i);
4710034     }
4710035     return NULL;
4710036 }

```

## 95.20.23 lib/string/strtok.c

»

Si veda la sezione 88.129.

```

4720001 #include <string.h>
4720002 //-----
4720003 char *
4720004 strtok (char *restrict string, const char *restrict delim)
4720005 {
4720006     static char *next = NULL;
4720007     size_t i = 0;
4720008     size_t j;
4720009     int found_token;
4720010     int found_delim;
4720011     //
4720012     // If the string received a the first parameter is a
4720013     // null pointer,
4720014     // the static pointer is used. But if it is already
4720015     // NULL,
4720016     // the scan cannot start.
4720017     //
4720018     if (string == NULL)
4720019     {
4720020         if (next == NULL)
4720021         {
4720022             return NULL;
4720023         }
4720024         else
4720025         {
4720026             string = next;
4720027         }
4720028     }
4720029     //
4720030     // If the string received as the first parameter is
4720031     // empty, the scan
4720032     // cannot start.
4720033     //

```



```

472034 if (string[0] == 0)
472035 {
472036     next = NULL;
472037     return NULL;
472038 }
472039 else
472040 {
472041     if (delim[0] == 0)
472042     {
472043         return string;
472044     }
472045 }
472046 //
472047 // Find the next token.
472048 //
472049 for (i = 0, found_token = 0, j = 0;
472050      string[i] != 0 && (!found_token); i++)
472051 {
472052     //
472053     // Look inside delimiters.
472054     //
472055     for (j = 0, found_delim = 0; delim[j] != 0; j++)
472056     {
472057         if (string[i] == delim[j])
472058         {
472059             found_delim = 1;
472060         }
472061     }
472062     //
472063     // If current character inside the string is not
472064     // a delimiter,
472065     // it is the start of a new token.
472066     //
472067     if (!found_delim)
472068     {
472069         found_token = 1;
472070         break;
472071     }
472072 }
472073 //
472074 // If a token was found, the pointer is updated.
472075 // If otherwise the token is not found, this means
472076 // that
472077 // there are no more.
472078 //
472079 if (found_token)
472080 {
472081     string += i;
472082 }
472083 else
472084 {
472085     next = NULL;
472086     return NULL;
472087 }
472088 //
472089 // Find the end of the token.
472090 //
472091 for (i = 0, found_delim = 0; string[i] != 0; i++)
472092 {
472093     for (j = 0; delim[j] != 0; j++)
472094     {
472095         if (string[i] == delim[j])
472096         {
472097             found_delim = 1;
472098             break;
472099         }
472100     }
472101     if (found_delim)
472102     {
472103         break;
472104     }
472105 }
472106 //
472107 // If a delimiter was found, the corresponding
472108 // character must be
472109 // reset to zero. If otherwise the string is
472110 // terminated, the
472111 // scan is terminated.
472112 //
472113 if (found_delim)
472114 {
472115     string[i] = 0;
472116     next = &string[i + 1];
472117 }
472118 else
472119 {
472120     next = NULL;

```

```

472021     }
472022     //
472023     // At this point, the current string represent the
472024     // token found.
472025     //
472026     return string;
472027 }

```

## 95.20.24 lib/string/strxfrm.c

Si veda la sezione 88.132.

```

473001 #include <string.h>
473002 //-----
473003 size_t
473004 strxfrm (char *restrict dst, const char *restrict org,
473005          size_t n)
473006 {
473007     size_t i;
473008     if (n == 0 && dst == NULL)
473009     {
473010         return strlen (org);
473011     }
473012     else
473013     {
473014         for (i = 0; i < n; i++)
473015         {
473016             dst[i] = org[i];
473017             if (org[i] == 0)
473018             {
473019                 break;
473020             }
473021         }
473022         return i;
473023     }
473024 }

```

## 95.21 os32: «lib/sys/os32.h»

Si veda la sezione 91.3.

```

474001 #ifndef _SYS_OS32_H
474002 #define _SYS_OS32_H    1
474003 //-----
474004 // This file contains all the declarations that don't
474005 // have a better place inside standard headers files.
474006 // Even declarations related to device numbers and
474007 // system calls is contained here.
474008 //-----
474009 #include <sys/types.h>
474010 #include <sys/stat.h>
474011 #include <sys/socket.h>
474012 #include <arpa/inet.h>
474013 #include <netinet/in.h>
474014 #include <stdint.h>
474015 #include <signal.h>
474016 #include <limits.h>
474017 #include <stdio.h>
474018 #include <stddef.h>
474019 #include <restrict.h>
474020 #include <stdarg.h>
474021 #include <termios.h>
474022 //-----
474023 typedef uint16_t h_port_t;    // Port number in host
474024                             // byte order.
474025 typedef uint32_t h_addr_t;    // IPv4 address in
474026                             // host byte order.
474027 //-----
474028 // Please remember that system calls should never be
474029 // used (called) inside the kernel code, because system
474030 // calls cannot be nested for the os32 simple
474031 // architecture!
474032 // If a particular function is necessary inside the
474033 // kernel, that usually is made by a system call, an
474034 // appropriate k_...() function must be
474035 // made, to avoid the problem.
474036 //-----
474037 // Device numbers.
474038 //-----
474039 #define DEV_UNDEFINED_MAJOR    ((dev_t) 0x00)
474040 #define DEV_UNDEFINED        ((dev_t) 0x0000)
474041 #define DEV_MEM_MAJOR        ((dev_t) 0x01)
474042 #define DEV_MEM              ((dev_t) 0x0101)
474043 #define DEV_NULL             ((dev_t) 0x0102)
474044 #define DEV_PORT             ((dev_t) 0x0103)
474045 #define DEV_ZERO             ((dev_t) 0x0104)
474046 #define DEV_TTY_MAJOR        ((dev_t) 0x02)

```

```

4740047 #define DEV_TTY ((dev_t) 0x0200)
4740048 //
4740049 #define DEV_KMEM_MAJOR ((dev_t) 0x04)
4740050 #define DEV_KMEM_PS ((dev_t) 0x0401)
4740051 #define DEV_KMEM_MMP ((dev_t) 0x0402)
4740052 #define DEV_KMEM_SB ((dev_t) 0x0403)
4740053 #define DEV_KMEM_INODE ((dev_t) 0x0404)
4740054 #define DEV_KMEM_FILE ((dev_t) 0x0405)
4740055 #define DEV_KMEM_ARP ((dev_t) 0x0406)
4740056 #define DEV_KMEM_NET ((dev_t) 0x0407)
4740057 #define DEV_KMEM_ROUTE ((dev_t) 0x0408)
4740058 //
4740059 #define DEV_CONSOLE_MAJOR ((dev_t) 0x05)
4740060 #define DEV_CONSOLE ((dev_t) 0x05FF)
4740061 #define DEV_CONSOLE0 ((dev_t) 0x0500)
4740062 #define DEV_CONSOLE1 ((dev_t) 0x0501)
4740063 #define DEV_CONSOLE2 ((dev_t) 0x0502)
4740064 #define DEV_CONSOLE3 ((dev_t) 0x0503)
4740065 #define DEV_CONSOLE4 ((dev_t) 0x0504)
4740066 //
4740067 #define DEV_DM_MAJOR ((dev_t) 0x08)
4740068 #define DEV_DM00 ((dev_t) 0x0800)
4740069 #define DEV_DM01 ((dev_t) 0x0801)
4740070 #define DEV_DM02 ((dev_t) 0x0802)
4740071 #define DEV_DM03 ((dev_t) 0x0803)
4740072 #define DEV_DM04 ((dev_t) 0x0804)
4740073 #define DEV_DM10 ((dev_t) 0x0810)
4740074 #define DEV_DM11 ((dev_t) 0x0811)
4740075 #define DEV_DM12 ((dev_t) 0x0812)
4740076 #define DEV_DM13 ((dev_t) 0x0813)
4740077 #define DEV_DM14 ((dev_t) 0x0814)
4740078 #define DEV_DM20 ((dev_t) 0x0820)
4740079 #define DEV_DM21 ((dev_t) 0x0821)
4740080 #define DEV_DM22 ((dev_t) 0x0822)
4740081 #define DEV_DM23 ((dev_t) 0x0823)
4740082 #define DEV_DM24 ((dev_t) 0x0824)
4740083 #define DEV_DM30 ((dev_t) 0x0830)
4740084 #define DEV_DM31 ((dev_t) 0x0831)
4740085 #define DEV_DM32 ((dev_t) 0x0832)
4740086 #define DEV_DM33 ((dev_t) 0x0833)
4740087 #define DEV_DM34 ((dev_t) 0x0834)
4740088 //
4740089 //-----
4740090 #define min(a, b) (a < b ? a : b)
4740091 #define max(a, b) (a > b ? a : b)
4740092 #define sizeof_array(x) (sizeof(x) / sizeof((x)[0]))
4740093 #define sizeof_field(t, f) (sizeof(((t*)0)->f))
4740094 //-----
4740095 #define INPUT_LINE_HIDDEN 0
4740096 #define INPUT_LINE_ECHO 1
4740097 //-----
4740098 #define MOUNT_DEFAULT 0 // Default mount
4740099 // options.
4740100 #define MOUNT_RO 1 // Read only mount
4740101 // option.
4740102 //-----
4740103 #define SYS_0 0 // Nothing to
4740104 // do.
4740105 #define SYS_CHDIR 1
4740106 #define SYS_CHMOD 2
4740107 #define SYS_CLOCK 3
4740108 #define SYS_CLOSE 4
4740109 #define SYS_EXEC 5
4740110 #define SYS_EXIT 6 // [1] see
4740111 // below.
4740112 #define SYS_FCHMOD 7
4740113 #define SYS_FORK 8
4740114 #define SYS_FSTAT 9
4740115 #define SYS_KILL 10
4740116 #define SYS_LSEEK 11
4740117 #define SYS_MKDIR 12
4740118 #define SYS_MKNOD 13
4740119 #define SYS_MOUNT 14
4740120 #define SYS_OPEN 15
4740121 #define SYS_PGRP 16
4740122 #define SYS_READ 17
4740123 #define SYS_SETEUID 18
4740124 #define SYS_SETUID 19
4740125 #define SYS_SIGNAL 20
4740126 #define SYS_SLEEP 21
4740127 #define SYS_STAT 22
4740128 #define SYS_TIME 23
4740129 #define SYS_UAREA 24
4740130 #define SYS_UMASK 25
4740131 #define SYS_UMOUNT 26
4740132 #define SYS_WAIT 27
4740133 #define SYS_WRITE 28

```

```

4740134 #define SYS_ZPCHAR 29 // [2]
4740135 #define SYS_ZPSTRING 30 // [2]
4740136 #define SYS_CHOWN 31
4740137 #define SYS_DUP 33
4740138 #define SYS_DUP2 34
4740139 #define SYS_LINK 35
4740140 #define SYS_UNLINK 36
4740141 #define SYS_FCNTL 37
4740142 #define SYS_STIME 38
4740143 #define SYS_FCHOWN 39
4740144 #define SYS_BRK 40
4740145 #define SYS_SBRK 41
4740146 #define SYS_PIPE 42
4740147 #define SYS_TCGETATTR 43
4740148 #define SYS_TCSETATTR 44
4740149 #define SYS_SETEGID 45
4740150 #define SYS_SETGID 46
4740151 #define SYS_SETJMP 47
4740152 #define SYS_LONGJMP 48
4740153 #define SYS_RECVFROM 49
4740154 #define SYS_SOCKET 50
4740155 #define SYS_CONNECT 51
4740156 #define SYS_SEND 52
4740157 #define SYS_IPCONFIG 53
4740158 #define SYS_ROUTEADD 54
4740159 #define SYS_ROUTEDEL 55
4740160 #define SYS_BIND 56
4740161 #define SYS_LISTEN 57
4740162 #define SYS_ACCEPT 58
4740163 //
4740164 // [1] The files 'crt0...' need to know the value used
4740165 // for the exit system call. If this value is
4740166 // modified, all the file 'crt0...' have also to be
4740167 // modified the same way.
4740168 //
4740169 // [2] These system calls were developed at the
4740170 // beginning, when no standard I/O was available.
4740171 // They are to be considered as a last resort for
4740172 // debugging purposes.
4740173 //
4740174 //-----
4740175 // The following values must be: 1, 2, 4, 8, 16, 32,...
4740176 // so that can be 'OR' combined.
4740177 //
4740178 #define WAKEUP_EVENT_SIGNAL 0x0001
4740179 #define WAKEUP_EVENT_TIMER 0x0002
4740180 #define WAKEUP_EVENT_DEV_READ 0x0004
4740181 #define WAKEUP_EVENT_DEV_WRITE 0x0008
4740182 #define WAKEUP_EVENT_PIPE_READ 0x0010
4740183 #define WAKEUP_EVENT_PIPE_WRITE 0x0020
4740184 #define WAKEUP_EVENT_SOCK_READ 0x0040
4740185 #define WAKEUP_EVENT_SOCK_WRITE 0x0080
4740186 //-----
4740187 typedef struct
4740188 {
4740189     int sfdn;
4740190     struct sockaddr addr;
4740191     socklen_t addrlen;
4740192     int fl_flags;
4740193     int ret;
4740194     int errno;
4740195     int errln;
4740196     char errfn[PATH_MAX];
4740197 } sysmsg_accept_t;
4740198 //-----
4740199 typedef struct
4740200 {
4740201     int sfdn;
4740202     struct sockaddr addr;
4740203     socklen_t addrlen;
4740204     int ret;
4740205     int errno;
4740206     int errln;
4740207     char errfn[PATH_MAX];
4740208 } sysmsg_bind_t;
4740209 //-----
4740210 typedef struct
4740211 {
4740212     void *address;
4740213     int ret;
4740214     int errno;
4740215     int errln;
4740216     char errfn[PATH_MAX];
4740217 } sysmsg_brk_t;
4740218 //-----
4740219 typedef struct
4740220 {

```

```

4740221 const char *path;
4740222 int ret;
4740223 int errno;
4740224 int errln;
4740225 char errfn[PATH_MAX];
4740226 } sysmsg_chdir_t;
4740227 //-----
4740228 typedef struct
4740229 {
4740230     const char *path;
4740231     mode_t mode;
4740232     int ret;
4740233     int errno;
4740234     int errln;
4740235     char errfn[PATH_MAX];
4740236 } sysmsg_chmod_t;
4740237 //-----
4740238 typedef struct
4740239 {
4740240     const char *path;
4740241     uid_t uid;
4740242     uid_t gid;
4740243     int ret;
4740244     int errno;
4740245     int errln;
4740246     char errfn[PATH_MAX];
4740247 } sysmsg_chown_t;
4740248 //-----
4740249 typedef struct
4740250 {
4740251     clock_t ret;
4740252 } sysmsg_clock_t;
4740253 //-----
4740254 typedef struct
4740255 {
4740256     int fdn;
4740257     int ret;
4740258     int errno;
4740259     int errln;
4740260     char errfn[PATH_MAX];
4740261 } sysmsg_close_t;
4740262 //-----
4740263 typedef struct
4740264 {
4740265     int sfdn;
4740266     struct sockaddr addr;
4740267     socklen_t addrlen;
4740268     int ret;
4740269     int errno;
4740270     int errln;
4740271     char errfn[PATH_MAX];
4740272 } sysmsg_connect_t;
4740273 //-----
4740274 typedef struct
4740275 {
4740276     int fdn_old;
4740277     int ret;
4740278     int errno;
4740279     int errln;
4740280     char errfn[PATH_MAX];
4740281 } sysmsg_dup_t;
4740282 //-----
4740283 typedef struct
4740284 {
4740285     int fdn_old;
4740286     int fdn_new;
4740287     int ret;
4740288     int errno;
4740289     int errln;
4740290     char errfn[PATH_MAX];
4740291 } sysmsg_dup2_t;
4740292 //-----
4740293 typedef struct
4740294 {
4740295     const char *path;
4740296     int argc;
4740297     int envc;
4740298     char arg_data[ARG_MAX / 2];
4740299     char env_data[ARG_MAX / 2];
4740300     uid_t uid;
4740301     uid_t euid;
4740302     int ret;
4740303     int errno;
4740304     int errln;
4740305     char errfn[PATH_MAX];
4740306 } sysmsg_exec_t;
4740307 //-----

```

```

4740308 typedef struct
4740309 {
4740310     int status;
4740311 } sysmsg_exit_t;
4740312 //-----
4740313 typedef struct
4740314 {
4740315     int fdn;
4740316     mode_t mode;
4740317     int ret;
4740318     int errno;
4740319     int errln;
4740320     char errfn[PATH_MAX];
4740321 } sysmsg_fchmod_t;
4740322 //-----
4740323 typedef struct
4740324 {
4740325     int fdn;
4740326     uid_t uid;
4740327     uid_t gid;
4740328     int ret;
4740329     int errno;
4740330     int errln;
4740331     char errfn[PATH_MAX];
4740332 } sysmsg_fchmod_t;
4740333 //-----
4740334 typedef struct
4740335 {
4740336     int fdn;
4740337     int cmd;
4740338     int arg;
4740339     int ret;
4740340     int errno;
4740341     int errln;
4740342     char errfn[PATH_MAX];
4740343 } sysmsg_fcntl_t;
4740344 //-----
4740345 typedef struct
4740346 {
4740347     pid_t ret;
4740348     int errno;
4740349     int errln;
4740350     char errfn[PATH_MAX];
4740351 } sysmsg_fork_t;
4740352 //-----
4740353 typedef struct
4740354 {
4740355     int fdn;
4740356     struct stat stat;
4740357     int ret;
4740358     int errno;
4740359     int errln;
4740360     char errfn[PATH_MAX];
4740361 } sysmsg_fstat_t;
4740362 //-----
4740363 typedef struct
4740364 {
4740365     int n;
4740366     in_addr_t address;
4740367     int m;
4740368     int ret;
4740369     int errno;
4740370     int errln;
4740371     char errfn[PATH_MAX];
4740372 } sysmsg_ipconfig_t;
4740373 //-----
4740374 typedef struct
4740375 {
4740376     void *env;
4740377     int ret;
4740378     //
4740379     // This structure is intentionally reduced.
4740380     //
4740381 } sysmsg_jmp_t;
4740382 //-----
4740383 typedef struct
4740384 {
4740385     pid_t pid;
4740386     int signal;
4740387     int ret;
4740388     int errno;
4740389     int errln;
4740390     char errfn[PATH_MAX];
4740391 } sysmsg_kill_t;
4740392 //-----
4740393 typedef struct
4740394 {

```

```

4740395 const char *path_old;
4740396 const char *path_new;
4740397 int ret;
4740398 int errno;
4740399 int errln;
4740400 char errfn[PATH_MAX];
4740401 } sysmsg_link_t;
4740402 //-----
4740403 typedef struct
4740404 {
4740405     int sfdn;
4740406     int backlog;
4740407     int ret;
4740408     int errno;
4740409     int errln;
4740410     char errfn[PATH_MAX];
4740411 } sysmsg_listen_t;
4740412 //-----
4740413 typedef struct
4740414 {
4740415     int fdn;
4740416     off_t offset;
4740417     int whence;
4740418     int ret;
4740419     int errno;
4740420     int errln;
4740421     char errfn[PATH_MAX];
4740422 } sysmsg_lseek_t;
4740423 //-----
4740424 typedef struct
4740425 {
4740426     const char *path;
4740427     mode_t mode;
4740428     int ret;
4740429     int errno;
4740430     int errln;
4740431     char errfn[PATH_MAX];
4740432 } sysmsg_mkdir_t;
4740433 //-----
4740434 typedef struct
4740435 {
4740436     const char *path;
4740437     mode_t mode;
4740438     dev_t device;
4740439     int ret;
4740440     int errno;
4740441     int errln;
4740442     char errfn[PATH_MAX];
4740443 } sysmsg_mknod_t;
4740444 //-----
4740445 typedef struct
4740446 {
4740447     const char *path_dev;
4740448     const char *path_mnt;
4740449     int options;
4740450     int ret;
4740451     int errno;
4740452     int errln;
4740453     char errfn[PATH_MAX];
4740454 } sysmsg_mount_t;
4740455 //-----
4740456 typedef struct
4740457 {
4740458     const char *path;
4740459     int flags;
4740460     mode_t mode;
4740461     int ret;
4740462     int errno;
4740463     int errln;
4740464     char errfn[PATH_MAX];
4740465 } sysmsg_open_t;
4740466 //-----
4740467 typedef struct
4740468 {
4740469     int pipefd[2];
4740470     int ret;
4740471     int errno;
4740472     int errln;
4740473     char errfn[PATH_MAX];
4740474 } sysmsg_pipe_t;
4740475 //-----
4740476 typedef struct
4740477 {
4740478     int fdn;
4740479     void *buffer;
4740480     size_t count;
4740481     int fl_flags;

```

```

4740482     ssize_t ret;
4740483     int errno;
4740484     int errln;
4740485     char errfn[PATH_MAX];
4740486 } sysmsg_read_t;
4740487 //-----
4740488 typedef struct
4740489 {
4740490     int sfdn;
4740491     void *buffer;
4740492     size_t count;
4740493     int flags;
4740494     void *addrfrom;
4740495     void *addrsz;
4740496     int fl_flags;
4740497     ssize_t ret;
4740498     int errno;
4740499     int errln;
4740500     char errfn[PATH_MAX];
4740501 } sysmsg_recvfrom_t;
4740502 //-----
4740503 typedef struct
4740504 {
4740505     in_addr_t destination;
4740506     int m;
4740507     in_addr_t router;
4740508     int device;
4740509     int ret;
4740510     int errno;
4740511     int errln;
4740512     char errfn[PATH_MAX];
4740513 } sysmsg_route_t;
4740514 //-----
4740515 typedef struct
4740516 {
4740517     intptr_t increment;
4740518     void *ret;
4740519     int errno;
4740520     int errln;
4740521     char errfn[PATH_MAX];
4740522 } sysmsg_sbrk_t;
4740523 //-----
4740524 typedef struct
4740525 {
4740526     int sfdn;
4740527     const void *buffer;
4740528     size_t count;
4740529     int flags;
4740530     ssize_t ret;
4740531     int errno;
4740532     int errln;
4740533     char errfn[PATH_MAX];
4740534 } sysmsg_send_t;
4740535 //-----
4740536 typedef struct
4740537 {
4740538     gid_t egid;
4740539     int ret;
4740540     int errno;
4740541     int errln;
4740542     char errfn[PATH_MAX];
4740543 } sysmsg_setegid_t;
4740544 //-----
4740545 typedef struct
4740546 {
4740547     uid_t euid;
4740548     int ret;
4740549     int errno;
4740550     int errln;
4740551     char errfn[PATH_MAX];
4740552 } sysmsg_seteuid_t;
4740553 //-----
4740554 typedef struct
4740555 {
4740556     gid_t gid;
4740557     gid_t egid;
4740558     gid_t sgid;
4740559     int ret;
4740560     int errno;
4740561     int errln;
4740562     char errfn[PATH_MAX];
4740563 } sysmsg_setgid_t;
4740564 //-----
4740565 typedef struct
4740566 {
4740567     uid_t uid;
4740568     uid_t euid;

```

```

4740569 uid_t suid;
4740570 int ret;
4740571 int errno;
4740572 int errln;
4740573 char errfn[PATH_MAX];
4740574 } sysmsg_setuid_t;
4740575 //-----
4740576 typedef struct
4740577 {
4740578     uintptr_t wrapper;
4740579     sighandler_t handler;
4740580     int signal;
4740581     sighandler_t ret;
4740582     int errno;
4740583     int errln;
4740584     char errfn[PATH_MAX];
4740585 } sysmsg_signal_t;
4740586 //-----
4740587 typedef struct
4740588 {
4740589     int family;
4740590     int type;
4740591     int protocol;
4740592     int ret;
4740593     int errno;
4740594     int errln;
4740595     char errfn[PATH_MAX];
4740596 } sysmsg_socket_t;
4740597 //-----
4740598 typedef struct
4740599 {
4740600     int events;
4740601     int signal;
4740602     unsigned int seconds;
4740603     time_t ret;
4740604 } sysmsg_sleep_t;
4740605 //-----
4740606 typedef struct
4740607 {
4740608     const char *path;
4740609     struct stat stat;
4740610     int ret;
4740611     int errno;
4740612     int errln;
4740613     char errfn[PATH_MAX];
4740614 } sysmsg_stat_t;
4740615 //-----
4740616 typedef struct
4740617 {
4740618     time_t ret;
4740619 } sysmsg_time_t;
4740620 //-----
4740621 typedef struct
4740622 {
4740623     time_t timer;
4740624     int ret;
4740625 } sysmsg_stime_t;
4740626 //-----
4740627 typedef struct
4740628 {
4740629     int fdn;
4740630     int action;
4740631     struct termios *attr;
4740632     int ret;
4740633     int errno;
4740634     int errln;
4740635     char errfn[PATH_MAX];
4740636 } sysmsg_tcatrt_t;
4740637 //-----
4740638 typedef struct
4740639 {
4740640     uid_t uid; // Read user ID.
4740641     uid_t euid; // Effective user ID.
4740642     uid_t suid; // Saved user ID.
4740643     gid_t gid; // Read group ID.
4740644     gid_t egid; // Effective group ID.
4740645     gid_t sgid; // Saved group ID.
4740646     pid_t pid; // Process ID.
4740647     pid_t ppid; // Parent PID.
4740648     pid_t pgrp; // Process group.
4740649     mode_t umask; // Access permission mask.
4740650     char *path_cwd;
4740651     size_t path_cwd_size; // Max path size.
4740652 } sysmsg_uarea_t;
4740653 //-----
4740654 typedef struct
4740655 {

```

```

4740656     mode_t umask;
4740657     mode_t ret;
4740658 } sysmsg_umask_t;
4740659 //-----
4740660 typedef struct
4740661 {
4740662     const char *path_mnt;
4740663     int ret;
4740664     int errno;
4740665     int errln;
4740666     char errfn[PATH_MAX];
4740667 } sysmsg_umount_t;
4740668 //-----
4740669 typedef struct
4740670 {
4740671     const char *path;
4740672     int ret;
4740673     int errno;
4740674     int errln;
4740675     char errfn[PATH_MAX];
4740676 } sysmsg_unlink_t;
4740677 //-----
4740678 typedef struct
4740679 {
4740680     int status;
4740681     pid_t ret;
4740682     int errno;
4740683     int errln;
4740684     char errfn[PATH_MAX];
4740685 } sysmsg_wait_t;
4740686 //-----
4740687 typedef struct
4740688 {
4740689     int fdn;
4740690     const void *buffer;
4740691     size_t count;
4740692     ssize_t ret;
4740693     int errno;
4740694     int errln;
4740695     char errfn[PATH_MAX];
4740696 } sysmsg_write_t;
4740697 //-----
4740698 typedef struct
4740699 {
4740700     char c;
4740701 } sysmsg_zpchar_t;
4740702 //-----
4740703 typedef struct
4740704 {
4740705     char string[BUFSIZ];
4740706 } sysmsg_zpstring_t;
4740707 //-----
4740708 void input_line (char *line, char *prompt, size_t size,
4740709                 int type);
4740710 int mount (const char *path_dev, const char *path_mnt,
4740711           int options);
4740712 int namep (const char *name, char *path, size_t size);
4740713 void sys (int syscallnr, void *message, size_t size);
4740714 int umount (const char *path_mnt);
4740715 void z_perror (const char *string);
4740716 int z_printf (const char *restrict format, ...);
4740717 int z_vprintf (const char *restrict format, va_list arg);
4740718 int ipconfig (int n, h_addr_t address, int m);
4740719 int routedel (h_addr_t destination, int m);
4740720 int routeadd (h_addr_t destination, int m,
4740721              h_addr_t router, int device);
4740722 //-----
4740723 #endif

```

95.21.1	lib/sys/os32/input_line.c	885
95.21.2	lib/sys/os32/ipconfig.c	887
95.21.3	lib/sys/os32/mount.c	888
95.21.4	lib/sys/os32/namep.c	888
95.21.5	lib/sys/os32/routeadd.c	890
95.21.6	lib/sys/os32/routedel.c	891
95.21.7	lib/sys/os32/sys.s	891
95.21.8	lib/sys/os32/umount.c	891
95.21.9	lib/sys/os32/z_perror.c	892
95.21.10	lib/sys/os32/z_printf.c	892
95.21.11	lib/sys/os32/z_vprintf.c	892

## 95.21.1 lib/sys/os32/input\_line.c

« Si veda la sezione 88.68.

```

4750001 #include <sys/os32.h>
4750002 #include <string.h>
4750003 #include <stdio.h>
4750004 #include <errno.h>
4750005 #include <unistd.h>
4750006 //-----
4750007 static int terminal_echo (struct termios *orig);
4750008 static int terminal_noecho (struct termios *orig);
4750009 static int terminal_restore (struct termios *orig);
4750010 //-----
4750011 void
4750012 input_line (char *line, char *prompt, size_t size, int type)
4750013 {
4750014     void *pstatus;
4750015     int i;
4750016     struct termios attr;
4750017     //
4750018     // Set terminal configuration.
4750019     //
4750020     if (type == INPUT_LINE_HIDDEN)
4750021     {
4750022         terminal_noecho (&attr);
4750023     }
4750024     else
4750025     {
4750026         terminal_echo (&attr);
4750027     }
4750028     //
4750029     if (prompt != NULL || strlen (prompt) > 0)
4750030     {
4750031         printf ("%s", prompt);
4750032     }
4750033     //
4750034     errno = 0;
4750035     pstatus = fgets (line, (int) size, stdin);
4750036     if (pstatus == NULL)
4750037     {
4750038         if (errno)
4750039         {
4750040             perror (NULL);
4750041         }
4750042         line[0] = 0;
4750043         //
4750044         // Reset terminal mode.
4750045         //
4750046         terminal_restore (&attr);
4750047         return;
4750048     }
4750049     //
4750050     // Find the last position and, if there is a new
4750051     // line code,
4750052     // replace it with zero. If the string is empty, a
4750053     // ^D was
4750054     // received.
4750055     //
4750056     i = strlen (line);
4750057     if (i > 0 && line[i - 1] == '\n')
4750058     {
4750059         line[i - 1] = '\0';
4750060     }
4750061     //
4750062     // Restore terminal mode.
4750063     //
4750064     terminal_restore (&attr);
4750065 }
4750066 //-----
4750067 static int
4750068 terminal_echo (struct termios *orig)
4750069 {
4750070     int status;
4750071     struct termios attr;
4750072     //
4750073     // Save previous.
4750074     //
4750075     //
4750076     status = tcgetattr (STDIN_FILENO, orig);
4750077     if (status < 0)
4750078     {
4750079         return (-1);
4750080     }
4750081     //
4750082     // Get again.
4750083     //
4750084     status = tcgetattr (STDIN_FILENO, &attr);
4750085     if (status < 0)

```

```

4750086     {
4750087         return (-1);
4750088     }
4750089     //
4750090     attr.c_iflag |= (BRKINT | ICRNL);
4750091     attr.c_iflag &= ~(IGNBRK | INLCR);
4750092     //
4750093     attr.c_lflag |=
4750094     (ECHO | ECHOE | ECHOK | ECHONL | ICANON | ISIG);
4750095     attr.c_lflag &= ~(IEXTEN);
4750096     //
4750097     status = tcsetattr (STDIN_FILENO, TCSANOW, &attr);
4750098     //
4750099     return (status);
4750100 }
4750101 //-----
4750102 static int
4750103 terminal_noecho (struct termios *orig)
4750104 {
4750105     int status;
4750106     struct termios attr;
4750107     //
4750108     // Save previous.
4750109     //
4750110     //
4750111     status = tcgetattr (STDIN_FILENO, orig);
4750112     if (status < 0)
4750113     {
4750114         return (-1);
4750115     }
4750116     //
4750117     // Get again.
4750118     //
4750119     status = tcgetattr (STDIN_FILENO, &attr);
4750120     if (status < 0)
4750121     {
4750122         return (-1);
4750123     }
4750124     //
4750125     attr.c_iflag |= (BRKINT | ICRNL);
4750126     attr.c_iflag &= ~(IGNBRK | INLCR);
4750127     //
4750128     attr.c_lflag |= (ICANON | ISIG);
4750129     attr.c_lflag &= ~(ECHO | IEXTEN);
4750130     //
4750131     status = tcsetattr (STDIN_FILENO, TCSANOW, &attr);
4750132     //
4750133     return (status);
4750134 }
4750135 //-----
4750136 static int
4750137 terminal_restore (struct termios *orig)
4750138 {
4750139     int status;
4750140     //
4750141     //
4750142     // For an unknown reason, when running with Bochs,
4750143     // before
4750144     // restoring the termios configuration, the previous
4750145     // one
4750146     // is to be read. Here, 'attr' is just a placeholder
4750147     // and
4750148     // the updated content is not used for anything
4750149     // else.
4750150     //
4750151     struct termios attr;
4750152     status = tcgetattr (STDIN_FILENO, &attr);
4750153     if (status < 0)
4750154     {
4750155         return (-1);
4750156     }
4750157     //
4750158     //
4750159     //
4750160     status = tcsetattr (STDIN_FILENO, TCSANOW, orig);
4750161     //
4750162     return (status);
4750163 }

```

## 95.21.2 lib/sys/os32/ipconfig.c

« Si veda la sezione 87.28.

```

4760001 #include <sys/os32.h>
4760002 #include <errno.h>
4760003 #include <string.h>
4760004 #include <stdio.h>

```

```

4760005 //-----
4760006 int
4760007 ipconfig (int n, in_addr_t address, int m)
4760008 {
4760009     sysmsg_ipconfig_t msg;
4760010     //
4760011     // Fill the message.
4760012     //
4760013     msg.n = n;
4760014     msg.address = address;
4760015     msg.m = m;
4760016     msg.ret = 0;
4760017     //
4760018     // Syscall.
4760019     //
4760020     sys (SYS_IPCONFIG, &msg, (sizeof msg));
4760021     //
4760022     // Check return value.
4760023     //
4760024     if (msg.ret < 0)
4760025     {
4760026         //
4760027         // Something wrong.
4760028         //
4760029         errno = msg.errno;
4760030         errln = msg.errln;
4760031         strncpy (errfn, msg.errfn, PATH_MAX);
4760032     }
4760033     //
4760034     // Return.
4760035     //
4760036     return (msg.ret);
4760037 }

```

### 95.21.3 lib/sys/os32/mount.c

« Si veda la sezione 87.36.

```

4770001 #include <sys/types.h>
4770002 #include <errno.h>
4770003 #include <sys/os32.h>
4770004 #include <stddef.h>
4770005 #include <string.h>
4770006 //-----
4770007 int
4770008 mount (const char *path_dev, const char *path_mnt,
4770009        int options)
4770010 {
4770011     sysmsg_mount_t msg;
4770012     //
4770013     msg.path_dev = path_dev;
4770014     msg.path_mnt = path_mnt;
4770015     msg.options = options;
4770016     msg.ret = 0;
4770017     msg.errno = 0;
4770018     //
4770019     sys (SYS_MOUNT, &msg, (sizeof msg));
4770020     //
4770021     errno = msg.errno;
4770022     errln = msg.errln;
4770023     strncpy (errfn, msg.errfn, PATH_MAX);
4770024     return (msg.ret);
4770025 }

```

### 95.21.4 lib/sys/os32/namep.c

« Si veda la sezione 88.85.

```

4780001 #include <sys/os32.h>
4780002 #include <stdlib.h>
4780003 #include <errno.h>
4780004 #include <unistd.h>
4780005 //-----
4780006 int
4780007 namep (const char *name, char *path, size_t size)
4780008 {
4780009     char command[PATH_MAX];
4780010     char *env_path;
4780011     int p; // Index used inside the path
4780012     // environment.
4780013     int c; // Index used inside the command
4780014     // string.
4780015     int status;
4780016     //
4780017     // Check for valid input.
4780018     //
4780019     if (name == NULL || name[0] == 0 || path == NULL

```

```

4780020     || name == path)
4780021     {
4780022         errset (EINVAL); // Invalid argument.
4780023         return (-1);
4780024     }
4780025     //
4780026     // Check if the original command contains at least a
4780027     // '/'. Otherwise
4780028     // a scan for the environment variable 'PATH' must
4780029     // be done.
4780030     //
4780031     if (strchr (name, '/') == NULL)
4780032     {
4780033         //
4780034         // Ok: no '/' there. Get the environment
4780035         // variable 'PATH'.
4780036         //
4780037         env_path = getenv ("PATH");
4780038         if (env_path == NULL)
4780039         {
4780040             //
4780041             // There is no 'PATH' environment value.
4780042             //
4780043             errset (ENOENT); // No such file or
4780044             // directory.
4780045             return (-1);
4780046         }
4780047         //
4780048         // Scan paths and try to find a file with that
4780049         // name.
4780050         //
4780051         for (p = 0; env_path[p] != 0;)
4780052         {
4780053             for (c = 0;
4780054                  c < (PATH_MAX - strlen (name) - 2) &&
4780055                     env_path[p] != 0 &&
4780056                     env_path[p] != ':'; c++, p++)
4780057             {
4780058                 command[c] = env_path[p];
4780059             }
4780060             //
4780061             // If the loop is ended because the command
4780062             // array does not
4780063             // have enough room for the full path, then
4780064             // must return an
4780065             // error.
4780066             //
4780067             if (env_path[p] != ':' && env_path[p] != 0)
4780068             {
4780069                 errset (ENAMETOOLONG); // Filename
4780070                 // too long.
4780071                 return (-1);
4780072             }
4780073             //
4780074             // The command array has enough space. At
4780075             // index 'c' must
4780076             // place a zero, to terminate current
4780077             // string.
4780078             //
4780079             command[c] = 0;
4780080             //
4780081             // Add the rest of the path.
4780082             //
4780083             strcat (command, "/");
4780084             strcat (command, name);
4780085             //
4780086             // Verify to have something with that full
4780087             // path name.
4780088             //
4780089             status = access (command, F_OK);
4780090             if (status == 0)
4780091             {
4780092                 //
4780093                 // Verify to have enough room inside the
4780094                 // destination
4780095                 // path.
4780096                 //
4780097                 if (strlen (command) >= size)
4780098                 {
4780099                     //
4780100                     // Sorry: too big. There must be
4780101                     // room also for
4780102                     // the string termination null
4780103                     // character.
4780104                     //
4780105                     errset (ENAMETOOLONG); // Filename
4780106                     // too long.

```

```

4780107         return (-1);
4780108     }
4780109     //
4780110     // Copy the path and return.
4780111     //
4780112     strncpy (path, command, size);
4780113     return (0);
4780114 }
4780115 //
4780116 // That path was not good: try again. But
4780117 // before returning
4780118 // to the external loop, must verify if 'p'
4780119 // is to be
4780120 // incremented, after a ':', because the
4780121 // external loop
4780122 // does not touch the index 'p',
4780123 //
4780124     if (env_path[p] == ':')
4780125     {
4780126         p++;
4780127     }
4780128 }
4780129 //
4780130 // At this point, there is no match with the
4780131 // paths.
4780132 //
4780133     errset (ENOENT); // No such file or directory.
4780134     return (-1);
4780135 }
4780136 //
4780137 // At this point, a path was given and the
4780138 // environment variable
4780139 // 'PATH' was not scanned. Just copy the same path.
4780140 // But must verify
4780141 // that the receiving path has enough room for it.
4780142 //
4780143     if (strlen (name) >= size)
4780144     {
4780145         //
4780146         // Sorry: too big.
4780147         //
4780148         errset (ENAMETOOLONG); // Filename too long.
4780149         return (-1);
4780150     }
4780151 //
4780152 // Ok: copy and return.
4780153 //
4780154     strncpy (path, name, size);
4780155     return (0);
4780156 }

```

## 95.21.5 lib/sys/os32/routeadd.c

« Si veda la sezione 87.42.

```

4790001 #include <sys/os32.h>
4790002 #include <errno.h>
4790003 #include <string.h>
4790004 #include <stdio.h>
4790005 //-----
4790006 int
4790007 routeadd (in_addr_t destination, int m,
4790008           in_addr_t router, int device)
4790009 {
4790010     sysmsg_route_t msg;
4790011     //
4790012     // Fill the message.
4790013     //
4790014     msg.destination = destination;
4790015     msg.m = m;
4790016     msg.router = router;
4790017     msg.device = device;
4790018     //
4790019     // Syscall.
4790020     //
4790021     sys (SYS_ROUTEADD, &msg, (sizeof msg));
4790022     //
4790023     // Check return value.
4790024     //
4790025     if (msg.ret < 0)
4790026     {
4790027         //
4790028         // Something wrong.
4790029         //
4790030         errno = msg.errno;
4790031         errln = msg.errln;
4790032         strncpy (errfn, msg.errfn, PATH_MAX);
4790033     }

```

```

4790033     }
4790034     //
4790035     // Return.
4790036     //
4790037     return (msg.ret);
4790038 }

```

## 95.21.6 lib/sys/os32/routedel.c

« Si veda la sezione 87.43.

```

4800001 #include <sys/os32.h>
4800002 #include <errno.h>
4800003 #include <string.h>
4800004 #include <stdio.h>
4800005 //-----
4800006 int
4800007 routedel (in_addr_t destination, int m)
4800008 {
4800009     sysmsg_route_t msg;
4800010     //
4800011     // Fill the message.
4800012     //
4800013     msg.destination = destination;
4800014     msg.m = m;
4800015     //
4800016     // Syscall.
4800017     //
4800018     sys (SYS_ROUTEDEL, &msg, (sizeof msg));
4800019     //
4800020     // Check return value.
4800021     //
4800022     if (msg.ret < 0)
4800023     {
4800024         //
4800025         // Something wrong.
4800026         //
4800027         errno = msg.errno;
4800028         errln = msg.errln;
4800029         strncpy (errfn, msg.errfn, PATH_MAX);
4800030     }
4800031     //
4800032     // Return.
4800033     //
4800034     return (msg.ret);
4800035 }

```

## 95.21.7 lib/sys/os32/sys.s

« Si veda la sezione 87.56.

```

4810001 .global sys
4810002 #-----
4810003 .text
4810004 #-----
4810005 # Call a system call.
4810006 #
4810007 # Please remember that system calls should never be
4810008 # used (called) inside the kernel code, because system
4810009 # calls cannot be nested for the os32 simple
4810010 # architecture!
4810011 # If a particular function is necessary inside the
4810012 # kernel, that usually is made by a system call, an
4810013 # appropriate k...() function must be made, to avoid
4810014 # the problem.
4810015 #
4810016 #-----
4810017 .align 4
4810018 sys:
4810019     int $128 # 0x80
4810020     ret

```

## 95.21.8 lib/sys/os32/umount.c

« Si veda la sezione 87.36.

```

4820001 #include <sys/types.h>
4820002 #include <errno.h>
4820003 #include <sys/os32.h>
4820004 #include <stddef.h>
4820005 #include <string.h>
4820006 //-----
4820007 int
4820008 umount (const char *path_mnt)
4820009 {
4820010     sysmsg_umount_t msg;
4820011     //

```



```

482012 msg.path_mnt = path_mnt;
482013 msg.ret = 0;
482014 msg.errno = 0;
482015 //
482016 sys (SYS_UMOUNT, &msg, (sizeof msg));
482017 //
482018 errno = msg.errno;
482019 errln = msg.errln;
482020 strncpy (errfn, msg.errfn, PATH_MAX);
482021 return (msg.ret);
482022 }

```

## 95.21.9 lib/sys/os32/z\_perror.c

« Si veda la sezione 87.65.

```

483001 #include <sys/os32.h>
483002 #include <errno.h>
483003 #include <stddef.h>
483004 #include <string.h>
483005 //-----
483006 void
483007 z_perror (const char *string)
483008 {
483009 //
483010 // If errno is zero, there is nothing to show.
483011 //
483012 if (errno == 0)
483013 {
483014 return;
483015 }
483016 //
483017 // Show the string if there is one.
483018 //
483019 if (string != NULL && strlen (string) > 0)
483020 {
483021 z_printf ("%s: ", string);
483022 }
483023 //
483024 // Show the translated error.
483025 //
483026 if (errfn[0] != 0 && errln != 0)
483027 {
483028 z_printf ("[%s:%u:%i] %s\n",
483029 errfn, errln, errno, strerror (errno));
483030 }
483031 else
483032 {
483033 z_printf ("[%i] %s\n", errno, strerror (errno));
483034 }
483035 }

```

## 95.21.10 lib/sys/os32/z\_printf.c

« Si veda la sezione 87.65.

```

484001 #include <sys/os32.h>
484002 #include <restrict.h>
484003 //-----
484004 int
484005 z_printf (const char *restrict format, ...)
484006 {
484007 va_list ap;
484008 va_start (ap, format);
484009 return z_vprintf (format, ap);
484010 }

```

## 95.21.11 lib/sys/os32/z\_vprintf.c

« Si veda la sezione 87.65.

```

485001 #include <sys/os32.h>
485002 #include <restrict.h>
485003 //-----
485004 int
485005 z_vprintf (const char *restrict format, va_list arg)
485006 {
485007 int ret;
485008 sysmsg_zpstring_t msg;
485009 msg.string[0] = 0;
485010 ret = vsprintf (msg.string, format, arg);
485011 sys (SYS_ZPSTRING, &msg, (sizeof msg));
485012 return ret;
485013 }

```

## 95.22 os32: «lib/sys/sa\_family\_t.h»

« Si veda la sezione 91.3.

```

486001 #ifndef _SYS_SA_FAMILY_T_H
486002 #define _SYS_SA_FAMILY_T_H 1
486003 //-----
486004 #include <inttypes.h>
486005 //-----
486006 typedef uint16_t sa_family_t; // Address family.
486007 //-----
486008 #endif

```

## 95.23 os32: «lib/sys/socket.h»

« Si veda la sezione 91.3.

```

487001 #ifndef _SYS_SOCKET_H
487002 #define _SYS_SOCKET_H 1
487003 //-----
487004 #include <stdint.h>
487005 #include <unistd.h>
487006 #include <sys/socklen_t.h>
487007 #include <sys/sa_family_t.h>
487008 //-----
487009 struct sockaddr
487010 {
487011 sa_family_t sa_family; // Address family.
487012 char sa_data[14]; // Socket address.
487013 };
487014 //
487015 //
487016 //
487017 struct sockaddr_storage
487018 {
487019 sa_family_t ss_family; // Socket storage
487020 // family.
487021 uint8_t ss_zero[14]; // Filler.
487022 };
487023 //
487024 //
487025 //
487026 #define SOCK_STREAM 1 // Byte-stream socket.
487027 #define SOCK_DGRAM 2 // Datagram socket.
487028 #define SOCK_RAW 3 // Raw protocol
487029 // interface.
487030 #define SOCK_SEQPACKET 5 // Sequenced-packet
487031 // socket.
487032 //
487033 // Protocol families:
487034 //
487035 #define PF_UNSPEC 0 // Unspecified.
487036 #define PF_UNIX 1 // Unix domain socket.
487037 #define PF_INET 2 // IPv4 protocol
487038 // family.
487039 #define PF_INET6 10 // IPv6 protocol
487040 // family.
487041 //
487042 // Address families.
487043 //
487044 #define AF_UNSPEC PF_UNSPEC // Unspecified.
487045 #define AF_UNIX PF_UNIX // Unix domain socket.
487046 #define AF_INET PF_INET // IPv4 address
487047 // family.
487048 #define AF_INET6 PF_INET6 // IPv6 address
487049 // family.
487050 //-----
487051 int accept (int sfdn, struct sockaddr *addr,
487052 socklen_t * addrln);
487053 int bind (int sfdn, const struct sockaddr *addr,
487054 socklen_t addrln);
487055 int connect (int sfdn, const struct sockaddr *addr,
487056 socklen_t addrln);
487057 int listen (int sfdn, int backlog);
487058 ssize_t send (int sfdn, const void *buffer,
487059 size_t count, int flags);
487060 ssize_t recvfrom (int sfdn, void *buffer, size_t count,
487061 int flags, struct sockaddr *addrfrom,
487062 socklen_t * addrln);
487063 int socket (int family, int type, int protocol);
487064 //
487065 #define recv(sfdn, buffer, count, flags) \
487066 recvfrom (sfdn, buffer, count, flags, NULL, NULL)
487067 //-----
487068 #endif

```

95.23.1	lib/sys/socket/accept.c	894
95.23.2	lib/sys/socket/bind.c	895
95.23.3	lib/sys/socket/connect.c	895
95.23.4	lib/sys/socket/listen.c	896
95.23.5	lib/sys/socket/recvfrom.c	896
95.23.6	lib/sys/socket/send.c	898
95.23.7	lib/sys/socket/socket.c	899

## 95.23.1 lib/sys/socket/accept.c

« Si veda la sezione 87.3.

```

488001 #include <sys/os32.h>
488002 #include <errno.h>
488003 #include <string.h>
488004 #include <stdio.h>
488005 #include <fcntl.h>
488006 //-----
488007 int
488008 accept (int sfdn, struct sockaddr *addr,
488009         socklen_t * addrlen)
488010 {
488011     sysmsg_accept_t msg;
488012     //
488013     // Fill the message.
488014     //
488015     msg.sfdn = sfdn;
488016     memset (&msg.addr, 0x00, sizeof (msg.addr));
488017     msg.addrlen = *addrlen;
488018     msg.fl_flags = 0; // Not necessary.
488019     msg.ret = 0;
488020     //
488021     // Syscall.
488022     //
488023     while (1)
488024     {
488025         sys (SYS_ACCEPT, &msg, (sizeof msg));
488026         //
488027         if (msg.ret < 0
488028             && (msg.errno == EAGAIN
488029                 || msg.errno == EWOULDBLOCK))
488030         {
488031             //
488032             // No request at the moment.
488033             //
488034             if (msg.fl_flags & O_NONBLOCK)
488035             {
488036                 //
488037                 // Don't block.
488038                 //
488039                 break;
488040             }
488041             else
488042             {
488043                 //
488044                 // Keep trying.
488045                 //
488046                 continue;
488047             }
488048         }
488049         else
488050         {
488051             break;
488052         }
488053     }
488054     //
488055     // Check return value.
488056     //
488057     if (msg.ret < 0)
488058     {
488059         //
488060         // Something wrong.
488061         //
488062         errno = msg.errno;
488063         errln = msg.errln;
488064         strncpy (errfn, msg.errfn, PATH_MAX);
488065     }
488066     else
488067     {
488068         //
488069         // Update the socket address and the address
488070         // length.
488071         //

```

```

488072         if (addrlen != NULL && addr != NULL && *addrlen > 0)
488073         {
488074             memcpy (addr, &msg.addr,
488075                     min (msg.addrlen, *addrlen));
488076             *addrlen = msg.addrlen;
488077         }
488078     }
488079     //
488080     // Return.
488081     //
488082     return (msg.ret);
488083 }

```

## 95.23.2 lib/sys/socket/bind.c

« Si veda la sezione 87.4.

```

489001 #include <sys/os32.h>
489002 #include <errno.h>
489003 #include <string.h>
489004 #include <stdio.h>
489005 //-----
489006 int
489007 bind (int sfdn, const struct sockaddr *addr,
489008       socklen_t addrlen)
489009 {
489010     sysmsg_bind_t msg;
489011     //
489012     // Fill the message.
489013     //
489014     msg.sfdn = sfdn;
489015     memcpy (&msg.addr, addr, (size_t) addrlen);
489016     msg.addrlen = addrlen;
489017     msg.ret = 0;
489018     //
489019     // Syscall.
489020     //
489021     sys (SYS_BIND, &msg, (sizeof msg));
489022     //
489023     // Check return value.
489024     //
489025     if (msg.ret < 0)
489026     {
489027         //
489028         // Something wrong.
489029         //
489030         errno = msg.errno;
489031         errln = msg.errln;
489032         strncpy (errfn, msg.errfn, PATH_MAX);
489033     }
489034     //
489035     // Return.
489036     //
489037     return (msg.ret);
489038 }

```

## 95.23.3 lib/sys/socket/connect.c

« Si veda la sezione 87.11.

```

490001 #include <sys/os32.h>
490002 #include <errno.h>
490003 #include <string.h>
490004 #include <stdio.h>
490005 //-----
490006 int
490007 connect (int sfdn, const struct sockaddr *addr,
490008         socklen_t addrlen)
490009 {
490010     sysmsg_connect_t msg;
490011     //
490012     // Fill the message.
490013     //
490014     msg.sfdn = sfdn;
490015     memcpy (&msg.addr, addr, (size_t) addrlen);
490016     msg.addrlen = addrlen;
490017     msg.ret = 0;
490018     //
490019     // Syscall.
490020     //
490021     while (1)
490022     {
490023         sys (SYS_CONNECT, &msg, (sizeof msg));
490024         //
490025         if (msg.ret < 0)
490026         {
490027             if (msg.errno == EINPROGRESS

```

```

490028     || msg.errno == EALREADY)
490029     {
490030         //
490031         // Loop until the connection is
490032         // established, or a
490033         // different error comes.
490034         //
490035         continue;
490036     }
490037     else
490038     {
490039         break;
490040     }
490041 }
490042 else
490043 {
490044     break;
490045 }
490046 }
490047 //
490048 // Check return value.
490049 //
490050 if (msg.ret < 0)
490051 {
490052     //
490053     // Something wrong.
490054     //
490055     errno = msg.errno;
490056     errln = msg.errln;
490057     strncpy (errfn, msg.errfn, PATH_MAX);
490058 }
490059 //
490060 // Return.
490061 //
490062 return (msg.ret);
490063 }

```

#### 95.23.4 lib/sys/socket/listen.c

« Si veda la sezione 87.31.

```

491001 #include <sys/os32.h>
491002 #include <errno.h>
491003 #include <string.h>
491004 #include <stdio.h>
491005 //-----
491006 int
491007 listen (int sfdn, int backlog)
491008 {
491009     sysmsg_listen_t msg;
491010     //
491011     // Fill the message.
491012     //
491013     msg.sfdn = sfdn;
491014     msg.backlog = backlog;
491015     msg.ret = 0;
491016     //
491017     // Syscall.
491018     //
491019     sys (SYS_LISTEN, &msg, (sizeof msg));
491020     //
491021     // Check return value.
491022     //
491023     if (msg.ret < 0)
491024     {
491025         //
491026         // Something wrong.
491027         //
491028         errno = msg.errno;
491029         errln = msg.errln;
491030         strncpy (errfn, msg.errfn, PATH_MAX);
491031     }
491032     //
491033     // Return.
491034     //
491035     return (msg.ret);
491036 }

```

#### 95.23.5 lib/sys/socket/recvfrom.c

« Si veda la sezione 87.40.

```

492001 #include <sys/os32.h>
492002 #include <errno.h>
492003 #include <string.h>
492004 #include <stdio.h>
492005 #include <fcntl.h>

```

```

492006 //-----
492007 ssize_t
492008 recvfrom (int sfdn, void *buffer, size_t count,
492009           int flags, struct sockaddr *addrfrom,
492010           socklen_t * addrlen)
492011 {
492012     sysmsg_recvfrom_t msg;
492013     //
492014     // Reduce size of read if necessary.
492015     //
492016     if (count > BUFSIZ)
492017     {
492018         count = BUFSIZ;
492019     }
492020     //
492021     // Fill the message.
492022     //
492023     msg.sfdn = sfdn;
492024     msg.buffer = buffer;
492025     msg.count = count;
492026     msg.flags = flags;
492027     msg.addrfrom = addrfrom;
492028     msg.addrlen = addrlen;
492029     msg.fl_flags = 0; // Not necessary.
492030     msg.ret = 0;
492031     //
492032     // Repeat syscall, until something is received or
492033     // end of file is
492034     // reached.
492035     //
492036     while (1)
492037     {
492038         sys (SYS_RECVFROM, &msg, (sizeof msg));
492039         if (msg.ret == 0)
492040         {
492041             //
492042             // Stream closed from the other side.
492043             //
492044             break;
492045         }
492046         if (msg.ret < 0
492047             && (msg.errno == EAGAIN
492048                 || msg.errno == EWOULDBLOCK))
492049         {
492050             //
492051             // No data at the moment.
492052             //
492053             if (msg.fl_flags & O_NONBLOCK)
492054             {
492055                 //
492056                 // Don't block.
492057                 //
492058                 break;
492059             }
492060             else
492061             {
492062                 //
492063                 // Keep trying.
492064                 //
492065                 continue;
492066             }
492067         }
492068         //
492069         // Otherwise, we have received something.
492070         //
492071         break;
492072     }
492073     //
492074     //
492075     //
492076     if (msg.ret < 0)
492077     {
492078         //
492079         // No valid read.
492080         //
492081         errno = msg.errno;
492082         errln = msg.errln;
492083         strncpy (errfn, msg.errfn, PATH_MAX);
492084         return (msg.ret);
492085     }
492086     //
492087     if (msg.ret > count)
492088     {
492089         //
492090         // A strange value was returned. Considering it
492091         // a read error.
492092         //

```

```

4920991     errset (EIO);    // I/O error.
4920994     return (-1);
4920995     }
4920996     //
4920997     // A valid read: return.
4920998     //
4920999     return (msg.ret);
4921000 }

```

## 95.23.6 lib/sys/socket/send.c

« Si veda la sezione 87.45.

```

4930001 #include <unistd.h>
4930002 #include <sys/os32.h>
4930003 #include <errno.h>
4930004 #include <string.h>
4930005 #include <stdio.h>
4930006 //-----
4930007 ssize_t
4930008 send (int sfdn, const void *buffer, size_t count, int flags)
4930009 {
4930010     sysmsg_send_t msg;
4930011     int retry = 3;
4930012     //
4930013     // Reduce size of write if necessary.
4930014     //
4930015     if (count > BUFSIZ)
4930016     {
4930017         count = BUFSIZ;
4930018     }
4930019     //
4930020     // Fill the message.
4930021     //
4930022     msg.sfdn = sfdn;
4930023     msg.buffer = buffer;
4930024     msg.count = count;
4930025     msg.flags = flags;
4930026     //
4930027     // Syscall.
4930028     //
4930029     for (; retry > 0; retry--)
4930030     {
4930031         sys (SYS_SEND, &msg, (sizeof msg));
4930032         //
4930033         // Check.
4930034         //
4930035         if ((msg.ret < 0) && (msg.errno == E_ARP_MISSING))
4930036         {
4930037             sleep (1);
4930038             continue;    // Retry.
4930039         }
4930040         else
4930041         {
4930042             break;
4930043         }
4930044     }
4930045     //
4930046     // Check the final result and return.
4930047     //
4930048     if (msg.ret < 0)
4930049     {
4930050         //
4930051         // No valid write.
4930052         //
4930053         errno = msg.errno;
4930054         errln = msg.errln;
4930055         strncpy (errfn, msg.errfn, PATH_MAX);
4930056         return (msg.ret);
4930057     }
4930058     //
4930059     if (msg.ret > count)
4930060     {
4930061         //
4930062         // A strange value was returned. Considering it
4930063         // a read error.
4930064         //
4930065         errset (EIO);    // I/O error.
4930066         return (-1);
4930067     }
4930068     //
4930069     // A valid write return.
4930070     //
4930071     return (msg.ret);
4930072 }

```

## 95.23.7 lib/sys/socket/socket.c

« Si veda la sezione 87.54.

```

4940001 #include <sys/os32.h>
4940002 #include <errno.h>
4940003 #include <string.h>
4940004 #include <stdio.h>
4940005 //-----
4940006 int
4940007 socket (int family, int type, int protocol)
4940008 {
4940009     sysmsg_socket_t msg;
4940010     //
4940011     // Fill the message.
4940012     //
4940013     msg.family = family;
4940014     msg.type = type;
4940015     msg.protocol = protocol;
4940016     msg.ret = 0;
4940017     //
4940018     // Syscall.
4940019     //
4940020     sys (SYS_SOCKET, &msg, (sizeof msg));
4940021     //
4940022     // Check return value.
4940023     //
4940024     if (msg.ret < 0)
4940025     {
4940026         //
4940027         // Something wrong.
4940028         //
4940029         errno = msg.errno;
4940030         errln = msg.errln;
4940031         strncpy (errfn, msg.errfn, PATH_MAX);
4940032     }
4940033     //
4940034     // Return.
4940035     //
4940036     return (msg.ret);
4940037 }

```

## 95.24 os32: «lib/sys/socklen\_t.h»

« Si veda la sezione 91.3.

```

4950001 #ifndef _SYS_SOCKLEN_T_H
4950002 #define _SYS_SOCKLEN_T_H    1
4950003 //-----
4950004 #include <stdint.h>
4950005 //-----
4950006 typedef uint32_t socklen_t;
4950007 //-----
4950008 #endif

```

## 95.25 os32: «lib/sys/stat.h»

« Si veda la sezione 91.3.

```

4960001 #ifndef _SYS_STAT_H
4960002 #define _SYS_STAT_H    1
4960003 //-----
4960004 #include <restrict.h>
4960005 #include <sys/types.h>    // dev_t
4960006                             // off_t
4960007                             // blkcnt_t
4960008                             // blksize_t
4960009                             // ino_t
4960010                             // mode_t
4960011                             // nlink_t
4960012                             // uid_t
4960013                             // gid_t
4960014                             // time_t
4960015 //-----
4960016 // File type.
4960017 //-----
4960018 #define S_IFMT    0170000    // File type mask.
4960019 //
4960020 #define S_IFBLK    0060000    // Block device file.
4960021 #define S_IFCHR    0020000    // Character device
4960022                             // file.
4960023 #define S_IFIFO    0010000    // Pipe (FIFO) file.
4960024 #define S_IFREG    0100000    // Regular file.
4960025 #define S_IFDIR    0040000    // Directory.
4960026 #define S_IFLNK    0120000    // Symbolic link.
4960027 #define S_IFSOCK    0140000    // Unix domain socket.

```

```

4960028 //-----
4960029 // Owner user access permissions.
4960030 //-----
4960031 #define S_IRWXU 0000700 // Owner user access
4960032 // permissions mask.
4960033 //
4960034 #define S_IRUSR 0000400 // Owner user read
4960035 // access permission.
4960036 #define S_IWUSR 0000200 // Owner user write
4960037 // access permission.
4960038 #define S_IXUSR 0000100 // Owner user
4960039 // execution or cross
4960040 // perm.
4960041 //-----
4960042 // Group owner access permissions.
4960043 //-----
4960044 #define S_IRWXG 0000070 // Owner group access
4960045 // permissions mask.
4960046 //
4960047 #define S_IRGRP 0000040 // Owner group read
4960048 // access permission.
4960049 #define S_IWGRP 0000020 // Owner group write
4960050 // access permission.
4960051 #define S_IXGRP 0000010 // Owner group
4960052 // execution or cross
4960053 // perm.
4960054 //-----
4960055 // Other users access permissions.
4960056 //-----
4960057 #define S_IRWXO 0000007 // Other users access
4960058 // permissions mask.
4960059 //
4960060 #define S_IROTH 0000004 // Other users read
4960061 // access permission.
4960062 #define S_IWOTH 0000002 // Other users write
4960063 // access permissions.
4960064 #define S_IXOTH 0000001 // Other users
4960065 // execution or cross
4960066 // perm.
4960067 //-----
4960068 // S-bit: in this case there is no mask to select all
4960069 // of them.
4960070 //-----
4960071 #define S_ISUID 0004000 // S-UID.
4960072 #define S_ISGID 0002000 // S-GID.
4960073 #define S_ISVTX 0001000 // Sticky.
4960074 //-----
4960075 // Macroinstructions to verify the type of file.
4960076 //-----
4960077 //
4960078 // Block device:
4960079 //
4960080 #define S_ISBLK(m) (((m) & S_IFMT) == S_IFBLK)
4960081 //
4960082 // Character device:
4960083 //
4960084 #define S_ISCHR(m) (((m) & S_IFMT) == S_IFCHR)
4960085 //
4960086 // FIFO.
4960087 //
4960088 #define S_ISFIFO(m) (((m) & S_IFMT) == S_ISFIFO)
4960089 //
4960090 // Regular file.
4960091 //
4960092 #define S_ISREG(m) (((m) & S_IFMT) == S_IFREG)
4960093 //
4960094 // Directory.
4960095 //
4960096 #define S_ISDIR(m) (((m) & S_IFMT) == S_IFDIR)
4960097 //
4960098 // Symbolic link.
4960099 //
4960100 #define S_ISLNK(m) (((m) & S_IFMT) == S_IFLNK)
4960101 //
4960102 // Socket (Unix domain socket).
4960103 //
4960104 #define S_ISSOCK(m) (((m) & S_IFMT) == S_ISSOCK)
4960105 //-----
4960106 // Structure 'stat'.
4960107 //-----
4960108 struct stat
4960109 {
4960110     dev_t st_dev; // Device containing the file.
4960111     ino_t st_ino; // File serial number (inode number).
4960112     mode_t st_mode; // File type and permissions.
4960113     nlink_t st_nlink; // Links to the file.
4960114     uid_t st_uid; // Owner user id.

```

```

4960115     gid_t st_gid; // Owner group id.
4960116     dev_t st_rdev; // Device number if it is a
4960117 // device file.
4960118     off_t st_size; // File size.
4960119     time_t st_atime; // Last access time.
4960120     time_t st_mtime; // Last modification time.
4960121     time_t st_ctime; // Last inode modification.
4960122     blksize_t st_blksize; // Block size for I/O
4960123 // operations.
4960124     blkcnt_t st_blocks; // File size / block size.
4960125 };
4960126 //-----
4960127 // Function prototypes.
4960128 //-----
4960129 int chmod (const char *path, mode_t mode);
4960130 int fchmod (int fdn, mode_t mode);
4960131 int fstat (int fdn, struct stat *buffer);
4960132 int lstat (const char *restrict path,
4960133           struct stat *restrict buffer);
4960134 int mkdir (const char *path, mode_t mode);
4960135 int mkfifo (const char *path, mode_t mode);
4960136 int mknod (const char *path, mode_t mode, dev_t dev);
4960137 int stat (const char *restrict path,
4960138         struct stat *restrict buffer);
4960139 mode_t umask (mode_t mask);
4960140
4960141 #endif // _SYS_STAT_H

```

95.25.1 lib/sys/stat/chmod.c ..... 901

95.25.2 lib/sys/stat/fchmod.c ..... 901

95.25.3 lib/sys/stat/fstat.c ..... 902

95.25.4 lib/sys/stat/mkdir.c ..... 902

95.25.5 lib/sys/stat/mknod.c ..... 903

95.25.6 lib/sys/stat/stat.c ..... 903

95.25.7 lib/sys/stat/umask.c ..... 903

95.25.1 lib/sys/stat/chmod.c

Si veda la sezione 87.7. »

```

4970001 #include <sys/stat.h>
4970002 #include <string.h>
4970003 #include <sys/os32.h>
4970004 #include <errno.h>
4970005 #include <limits.h>
4970006 //-----
4970007 int
4970008 chmod (const char *path, mode_t mode)
4970009 {
4970010     sysmsg_chmod_t msg;
4970011     //
4970012     msg.path = path;
4970013     msg.mode = mode;
4970014     //
4970015     sys (SYS_CHMOD, &msg, (sizeof msg));
4970016     //
4970017     errno = msg.errno;
4970018     errln = msg.errln;
4970019     strncpy (errfn, msg.errfn, PATH_MAX);
4970020     return (msg.ret);
4970021 }

```

95.25.2 lib/sys/stat/fchmod.c »

Si veda la sezione 87.7.

```

4980001 #include <sys/stat.h>
4980002 #include <string.h>
4980003 #include <sys/os32.h>
4980004 #include <errno.h>
4980005 #include <limits.h>
4980006 //-----
4980007 int
4980008 fchmod (int fdn, mode_t mode)
4980009 {
4980010     sysmsg_fchmod_t msg;
4980011     //
4980012     msg.fdn = fdn;
4980013     msg.mode = mode;
4980014     //
4980015     sys (SYS_FCHMOD, &msg, (sizeof msg));
4980016     //

```

```

498017     errno = msg.errno;
498018     errln = msg.errln;
498019     strncpy (errfn, msg.errfn, PATH_MAX);
498020     return (msg.ret);
498021 }

```

### 95.25.3 lib/sys/stat/fstat.c

« Si veda la sezione 87.55.

```

499001 #include <unistd.h>
499002 #include <errno.h>
499003 #include <sys/os32.h>
499004 #include <string.h>
499005 //-----
499006 int
499007 fstat (int fdn, struct stat *buffer)
499008 {
499009     sysmsg_fstat_t msg;
499010     //
499011     msg.fdn = fdn;
499012     msg.stat.st_dev = buffer->st_dev;
499013     msg.stat.st_ino = buffer->st_ino;
499014     msg.stat.st_mode = buffer->st_mode;
499015     msg.stat.st_nlink = buffer->st_nlink;
499016     msg.stat.st_uid = buffer->st_uid;
499017     msg.stat.st_gid = buffer->st_gid;
499018     msg.stat.st_rdev = buffer->st_rdev;
499019     msg.stat.st_size = buffer->st_size;
499020     msg.stat.st_atime = buffer->st_atime;
499021     msg.stat.st_mtime = buffer->st_mtime;
499022     msg.stat.st_ctime = buffer->st_ctime;
499023     msg.stat.st_blksize = buffer->st_blksize;
499024     msg.stat.st_blocks = buffer->st_blocks;
499025     //
499026     sys (SYS_FSTAT, &msg, (sizeof msg));
499027     //
499028     buffer->st_dev = msg.stat.st_dev;
499029     buffer->st_ino = msg.stat.st_ino;
499030     buffer->st_mode = msg.stat.st_mode;
499031     buffer->st_nlink = msg.stat.st_nlink;
499032     buffer->st_uid = msg.stat.st_uid;
499033     buffer->st_gid = msg.stat.st_gid;
499034     buffer->st_rdev = msg.stat.st_rdev;
499035     buffer->st_size = msg.stat.st_size;
499036     buffer->st_atime = msg.stat.st_atime;
499037     buffer->st_mtime = msg.stat.st_mtime;
499038     buffer->st_ctime = msg.stat.st_ctime;
499039     buffer->st_blksize = msg.stat.st_blksize;
499040     buffer->st_blocks = msg.stat.st_blocks;
499041     //
499042     errno = msg.errno;
499043     errln = msg.errln;
499044     strncpy (errfn, msg.errfn, PATH_MAX);
499045     return (msg.ret);
499046 }

```

### 95.25.4 lib/sys/stat/mkdir.c

« Si veda la sezione 87.34.

```

500001 #include <sys/stat.h>
500002 #include <string.h>
500003 #include <sys/os32.h>
500004 #include <errno.h>
500005 #include <limits.h>
500006 //-----
500007 int
500008 mkdir (const char *path, mode_t mode)
500009 {
500010     sysmsg_mkdir_t msg;
500011     //
500012     msg.path = path;
500013     msg.mode = mode;
500014     //
500015     sys (SYS_MKDIR, &msg, (sizeof msg));
500016     //
500017     errno = msg.errno;
500018     errln = msg.errln;
500019     strncpy (errfn, msg.errfn, PATH_MAX);
500020     return (msg.ret);
500021 }

```

### 95.25.5 lib/sys/stat/mknod.c

« Si veda la sezione 87.35.

```

501001 #include <unistd.h>
501002 #include <errno.h>
501003 #include <sys/os32.h>
501004 #include <string.h>
501005 //-----
501006 int
501007 mknod (const char *path, mode_t mode, dev_t device)
501008 {
501009     sysmsg_mknod_t msg;
501010     //
501011     msg.path = path;
501012     msg.mode = mode;
501013     msg.device = device;
501014     //
501015     sys (SYS_MKNOD, &msg, (sizeof msg));
501016     //
501017     errno = msg.errno;
501018     errln = msg.errln;
501019     strncpy (errfn, msg.errfn, PATH_MAX);
501020     return (msg.ret);
501021 }

```

### 95.25.6 lib/sys/stat/stat.c

« Si veda la sezione 87.55.

```

502001 #include <unistd.h>
502002 #include <errno.h>
502003 #include <sys/os32.h>
502004 #include <string.h>
502005 //-----
502006 int
502007 stat (const char *path, struct stat *buffer)
502008 {
502009     sysmsg_stat_t msg;
502010     //
502011     msg.path = path;
502012     //
502013     msg.stat.st_dev = buffer->st_dev;
502014     msg.stat.st_ino = buffer->st_ino;
502015     msg.stat.st_mode = buffer->st_mode;
502016     msg.stat.st_nlink = buffer->st_nlink;
502017     msg.stat.st_uid = buffer->st_uid;
502018     msg.stat.st_gid = buffer->st_gid;
502019     msg.stat.st_rdev = buffer->st_rdev;
502020     msg.stat.st_size = buffer->st_size;
502021     msg.stat.st_atime = buffer->st_atime;
502022     msg.stat.st_mtime = buffer->st_mtime;
502023     msg.stat.st_ctime = buffer->st_ctime;
502024     msg.stat.st_blksize = buffer->st_blksize;
502025     msg.stat.st_blocks = buffer->st_blocks;
502026     //
502027     sys (SYS_STAT, &msg, (sizeof msg));
502028     //
502029     buffer->st_dev = msg.stat.st_dev;
502030     buffer->st_ino = msg.stat.st_ino;
502031     buffer->st_mode = msg.stat.st_mode;
502032     buffer->st_nlink = msg.stat.st_nlink;
502033     buffer->st_uid = msg.stat.st_uid;
502034     buffer->st_gid = msg.stat.st_gid;
502035     buffer->st_rdev = msg.stat.st_rdev;
502036     buffer->st_size = msg.stat.st_size;
502037     buffer->st_atime = msg.stat.st_atime;
502038     buffer->st_mtime = msg.stat.st_mtime;
502039     buffer->st_ctime = msg.stat.st_ctime;
502040     buffer->st_blksize = msg.stat.st_blksize;
502041     buffer->st_blocks = msg.stat.st_blocks;
502042     //
502043     errno = msg.errno;
502044     errln = msg.errln;
502045     strncpy (errfn, msg.errfn, PATH_MAX);
502046     return (msg.ret);
502047 }

```

### 95.25.7 lib/sys/stat/umask.c

« Si veda la sezione 87.60.

```

503001 #include <sys/stat.h>
503002 #include <string.h>
503003 #include <sys/os32.h>
503004 #include <errno.h>
503005 #include <limits.h>
503006 //-----

```

```

5030007 mode_t
5030008 umask (mode_t mask)
5030009 {
5030010     sysmsg_umask_t msg;
5030011     msg.umask = mask;
5030012     sys (SYS_UMASK, &msg, (sizeof msg));
5030013     return (msg.ret);
5030014 }

```

## 95.26 os32: «lib/sys/types.h»

« Si veda la sezione 91.3.

```

5040001 #ifndef _SYS_TYPES_H
5040002 #define _SYS_TYPES_H 1
5040003 //-----
5040004 #include <clock_t.h>
5040005 #include <time_t.h>
5040006 #include <size_t.h>
5040007 //-----
5040008 typedef long int blkcnt_t;
5040009 typedef long int blksize_t;
5040010 typedef uint16_t dev_t; // Traditional device size.
5040011 typedef unsigned int id_t;
5040012 typedef unsigned int gid_t;
5040013 typedef unsigned int uid_t;
5040014 typedef uint16_t ino_t; // Minix 1 file system inode
5040015 // size.
5040016 typedef uint16_t mode_t; // Minix 1 file system
5040017 // mode size.
5040018 typedef unsigned int nlink_t;
5040019 typedef long long int off_t;
5040020 typedef int pid_t;
5040021 typedef unsigned long int pthread_t;
5040022 typedef int ssize_t;
5040023 //-----
5040024 // Common extentions.
5040025 //
5040026 dev_t makedev (int major, int minor);
5040027 int major (dev_t device);
5040028 int minor (dev_t device);
5040029 //-----
5040030 #endif

```

95.26.1 lib/sys/types/major.c ..... 904

95.26.2 lib/sys/types/makedev.c ..... 904

95.26.3 lib/sys/types/minor.c ..... 904

## 95.26.1 lib/sys/types/major.c

« Si veda la sezione 88.75.

```

5050001 #include <sys/types.h>
5050002 //-----
5050003 int
5050004 major (dev_t device)
5050005 {
5050006     return ((int) (device / 256));
5050007 }

```

## 95.26.2 lib/sys/types/makedev.c

« Si veda la sezione 88.75.

```

5060001 #include <sys/types.h>
5060002 //-----
5060003 dev_t
5060004 makedev (int major, int minor)
5060005 {
5060006     return ((dev_t) (major * 256 + minor));
5060007 }

```

## 95.26.3 lib/sys/types/minor.c

« Si veda la sezione 88.75.

```

5070001 #include <sys/types.h>
5070002 //-----
5070003 int
5070004 minor (dev_t device)
5070005 {
5070006     return ((dev_t) (device & 0x00FF));
5070007 }

```

## 95.27 os32: «lib/sys/wait.h»

« Si veda la sezione 91.3.

```

5080001 #ifndef _SYS_WAIT_H
5080002 #define _SYS_WAIT_H 1
5080003 //-----
5080004 #include <sys/types.h>
5080005 //-----
5080006 //-----
5080007 pid_t wait (int *status);
5080008 //-----
5080009 //-----
5080010 #endif

```

95.27.1 lib/sys/wait/wait.c ..... 905

## 95.27.1 lib/sys/wait/wait.c

« Si veda la sezione 87.63.

```

5090001 #include <sys/types.h>
5090002 #include <errno.h>
5090003 #include <sys/os32.h>
5090004 #include <stddef.h>
5090005 #include <string.h>
5090006 //-----
5090007 pid_t
5090008 wait (int *status)
5090009 {
5090010     sysmsg_wait_t msg;
5090011     msg.ret = 0;
5090012     msg.errno = 0;
5090013     msg.status = 0;
5090014     while (msg.ret == 0)
5090015     {
5090016         //
5090017         // Loop as long as there are children, an none
5090018         // is dead.
5090019         //
5090020         sys (SYS_WAIT, &msg, (sizeof msg));
5090021     }
5090022     errno = msg.errno;
5090023     errln = msg.errln;
5090024     strncpy (errfn, msg.errfn, PATH_MAX);
5090025     //
5090026     if (status != NULL)
5090027     {
5090028         //
5090029         // Only the low eight bits are returned.
5090030         //
5090031         *status = (msg.status & 0x00FF);
5090032     }
5090033     return (msg.ret);
5090034 }

```

## 95.28 os32: «lib/termios.h»

« Si veda la sezione 87.58.

```

5100001 #ifndef _TERMIOS_H
5100002 #define _TERMIOS_H 1
5100003 //-----
5100004 #include <stdint.h>
5100005 //-----
5100006 typedef uint16_t tcfld_t;
5100007 typedef unsigned char cc_t;
5100008 //-----
5100009 #define NCCS 11 // 'c_cc[]' size.
5100010 //
5100011 struct termios
5100012 {
5100013     tcfld_t c_iflag;
5100014     tcfld_t c_oflag;
5100015     tcfld_t c_cflag;
5100016     tcfld_t c_lflag;
5100017     cc_t c_cc[NCCS];
5100018 };
5100019 //
5100020 // Subscript names for 'c_cc[]' array, inside the
5100021 // 'termios' structure:
5100022 //
5100023 #define VEOF 0 // EOF character
5100024 #define VEOL 1 // EOL character
5100025 #define VERASE 2 // ERASE character
5100026 #define VINTR 3 // INTR character
5100027 #define VKILL 4 // KILL character
5100028 #define VMIN 5 // MIN value

```

```

510029 #define VQUIT 6 // QUIT character
510030 #define VSTART 7 // START character
510031 #define VSTOP 8 // STOP character
510032 #define VSUSP 9 // SUSP character
510033 #define VTIME 10 // TIME value
510034 //
510035 // Input modes, for 'c_iflag' inside the 'termios'
510036 // structure:
510037 //
510038 #define BRKINT 1 // signal interrupt on break
510039 #define ICRNL 2 // map CR to NL on input
510040 #define IGNBRK 4 // ignore break condition
510041 #define IGNCR 8 // ignore CR
510042 #define IGNPAR 16 // ignore characters with
510043 // parity errors
510044 #define INLCR 32 // map NL to CR on input
510045 #define INPCK 64 // enable input parity check
510046 #define ISTRIP 128 // strip off eighth bit
510047 #define IXOFF 256 // enable start/stop input
510048 // control
510049 #define IXON 512 // enable start/stop output
510050 // control
510051 #define PARMRK 1024 // mark parity errors
510052 //
510053 // Output modes, for 'c_oflag' inside the 'termios'
510054 // structure:
510055 //
510056 #define OPOST 1 // post-process output
510057 //
510058 // Control modes, for 'c_cflag' inside the 'termios'
510059 // structure:
510060 // not implemented.
510061 //
510062 //
510063 // Local modes, for 'c_iflag' inside the 'termios'
510064 // structure:
510065 //
510066 #define ECHO 1 // enable echo
510067 #define ECHOE 2 // echo erase character as
510068 // backspace
510069 #define ECHOK 4 // echo KILL
510070 #define ECHONL 8 // echo NL
510071 #define ICANON 16 // canonical input mode
510072 #define IEXTEN 32 // extended input mode
510073 #define ISIG 64 // enable signals
510074 #define NOFLSH 128 // disable flush after
510075 // interrupt or quit
510076 #define TOSTOP 256 // send SIGTTOU for background
510077 // output
510078 //-----
510079 // Optional action for use with 'tcsetattr()':
510080 //
510081 #define TCSANOW 1 // change attributes
510082 // immediately
510083 #define TCSADRAIN 2 // change attributes when
510084 // output has drained
510085 #define TCSAFLUSH 3 // change attributes when
510086 // output has drained,
510087 // and also flush pending
510088 // input
510089 //-----
510090 int tcgetattr (int fdn, struct termios *termios_p);
510091 int tcsetattr (int fdn, int action,
510092 // struct termios *termios_p);
510093 //-----
510094 #endif

```

95.28.1 lib/termios/tcgetattr.c .....906

95.28.2 lib/termios/tcsetattr.c ..... 907

### 95.28.1 lib/termios/tcgetattr.c

« Si veda la sezione 87.58.

```

510001 #include <termios.h>
510002 #include <sys/os32.h>
510003 #include <errno.h>
510004 //-----
510005 #define DEBUG 0
510006 //-----
510007 int
510008 tcgetattr (int fdn, struct termios *termios_p)
510009 {
510010     sysmsg_tcattr_t msg;
510011     msg.fdn = fdn;
510012     msg.attr = termios_p;

```

```

510013     sys (SYS_TCGETATTR, &msg, (sizeof msg));
510014     errno = msg.errno;
510015     errln = msg.errln;
510016     strncpy (errfn, msg.errfn, PATH_MAX);
510017     return (msg.ret);
510018 }

```

### 95.28.2 lib/termios/tcsetattr.c

« Si veda la sezione 87.58.

```

512001 #include <termios.h>
512002 #include <sys/os32.h>
512003 #include <errno.h>
512004 //-----
512005 #define DEBUG 0
512006 //-----
512007 int
512008 tcsetattr (int fdn, int action, struct termios *termios_p)
512009 {
512010     sysmsg_tcattr_t msg;
512011     msg.fdn = fdn;
512012     msg.action = action;
512013     msg.attr = termios_p;
512014     sys (SYS_TCSETATTR, &msg, (sizeof msg));
512015     errno = msg.errno;
512016     errln = msg.errln;
512017     strncpy (errfn, msg.errfn, PATH_MAX);
512018     return (msg.ret);
512019 }

```

### 95.29 os32: «lib/time.h»

« Si veda la sezione 91.3.

```

513001 #ifndef _TIME_H
513002 #define _TIME_H 1
513003 //-----
513004 #include <restrict.h>
513005 #include <size_t.h>
513006 #include <time_t.h>
513007 #include <clock_t.h>
513008 #include <NULL.h>
513009 #include <stdint.h>
513010 //-----
513011 #define CLOCKS_PER_SEC ((clock_t) 100)
513012 //-----
513013 struct tm
513014 {
513015     int tm_sec;
513016     int tm_min;
513017     int tm_hour;
513018     int tm_mday;
513019     int tm_mon;
513020     int tm_year;
513021     int tm_wday;
513022     int tm_yday;
513023     int tm_isdst;
513024 };
513025 //-----
513026 clock_t clock (void);
513027 time_t time (time_t * timer);
513028 int stime (time_t * timer);
513029 double difftime (time_t time1, time_t time0);
513030 time_t mktime (const struct tm *timeptr);
513031 struct tm *gmtime (const time_t * timer);
513032 struct tm *localtime (const time_t * timer);
513033 char *asctime (const struct tm *timeptr);
513034 char *ctime (const time_t * timer);
513035 size_t strftime (char *restrict s, size_t maxsize,
513036 // const char *restrict format,
513037 // const struct tm *restrict timeptr);
513038 //-----
513039 #define difftime(t1,t0) ((double)((t1)-(t0)))
513040 #define ctime(t) (asctime (localtime (t)))
513041 #define localtime(t) (gmtime (t))
513042 //-----
513043 #endif

```

95.29.1 lib/time/asctime.c ..... 908

95.29.2 lib/time/clock.c ..... 909

95.29.3 lib/time/gmtime.c ..... 909

95.29.4 lib/time/mktime.c ..... 911



95.29.5	lib/time/stime.c	913
95.29.6	lib/time/time.c	913

## 95.29.1 lib/time/asctime.c

&lt;&lt;

Si veda la sezione 88.15.

```

5140001 #include <time.h>
5140002 #include <string.h>
5140003 #include <stdio.h>
5140004 -----
5140005 char *
5140006 asctime (const struct tm *timeptr)
5140007 {
5140008     static char time_string[25]; // 'Sun Jan 30
5140009     // 24:00:00 2111'
5140010     //
5140011     // Check argument.
5140012     //
5140013     if (timeptr == NULL)
5140014     {
5140015         return (NULL);
5140016     }
5140017     //
5140018     // Set week day.
5140019     //
5140020     switch (timeptr->tm_wday)
5140021     {
5140022     case 0:
5140023         strcpy (&time_string[0], "Sun");
5140024         break;
5140025     case 1:
5140026         strcpy (&time_string[0], "Mon");
5140027         break;
5140028     case 2:
5140029         strcpy (&time_string[0], "Tue");
5140030         break;
5140031     case 3:
5140032         strcpy (&time_string[0], "Wed");
5140033         break;
5140034     case 4:
5140035         strcpy (&time_string[0], "Thu");
5140036         break;
5140037     case 5:
5140038         strcpy (&time_string[0], "Fri");
5140039         break;
5140040     case 6:
5140041         strcpy (&time_string[0], "Sat");
5140042         break;
5140043     default:
5140044         strcpy (&time_string[0], "Err");
5140045     }
5140046     //
5140047     // Set month.
5140048     //
5140049     switch (timeptr->tm_mon)
5140050     {
5140051     case 1:
5140052         strcpy (&time_string[3], " Jan");
5140053         break;
5140054     case 2:
5140055         strcpy (&time_string[3], " Feb");
5140056         break;
5140057     case 3:
5140058         strcpy (&time_string[3], " Mar");
5140059         break;
5140060     case 4:
5140061         strcpy (&time_string[3], " Apr");
5140062         break;
5140063     case 5:
5140064         strcpy (&time_string[3], " May");
5140065         break;
5140066     case 6:
5140067         strcpy (&time_string[3], " Jun");
5140068         break;
5140069     case 7:
5140070         strcpy (&time_string[3], " Jul");
5140071         break;
5140072     case 8:
5140073         strcpy (&time_string[3], " Aug");
5140074         break;
5140075     case 9:
5140076         strcpy (&time_string[3], " Sep");
5140077         break;
5140078     case 10:
5140079         strcpy (&time_string[3], " Oct");
5140080         break;

```

```

5140081     case 11:
5140082         strcpy (&time_string[3], " Nov");
5140083         break;
5140084     case 12:
5140085         strcpy (&time_string[3], " Dec");
5140086         break;
5140087     default:
5140088         strcpy (&time_string[3], " Err");
5140089     }
5140090     //
5140091     // Set day of month, hour, minute, second and year.
5140092     //
5140093     sprintf (&time_string[7], " %2i %2i:%2i:%2i %4i",
5140094             timeptr->tm_mday, timeptr->tm_hour,
5140095             timeptr->tm_min, timeptr->tm_sec,
5140096             timeptr->tm_year);
5140097     //
5140098     //
5140099     //
5140100     return (&time_string[0]);
5140101 }

```

## 95.29.2 lib/time/clock.c

&lt;&lt;

Si veda la sezione 87.9.

```

5150001 #include <time.h>
5150002 #include <sys/os32.h>
5150003 -----
5150004 clock_t
5150005 clock (void)
5150006 {
5150007     sysmsg_clock_t msg;
5150008     msg.ret = 0;
5150009     sys (SYS_CLOCK, &msg, (sizeof msg));
5150010     return (msg.ret);
5150011 }

```

## 95.29.3 lib/time/gmtime.c

&lt;&lt;

Si veda la sezione 88.15.

```

5160001 #include <time.h>
5160002 -----
5160003 static int leap_year (int year);
5160004 -----
5160005 struct tm *
5160006 gmtime (const time_t * timer)
5160007 {
5160008     static struct tm tms;
5160009     int loop;
5160010     unsigned int remainder;
5160011     unsigned int days;
5160012     //
5160013     // Check argument.
5160014     //
5160015     if (timer == NULL)
5160016     {
5160017         return (NULL);
5160018     }
5160019     //
5160020     // Days since epoch. There are 86400 seconds per
5160021     // day.
5160022     // At the moment, the field 'tm_yday' will contain
5160023     // all days since epoch.
5160024     //
5160025     days = *timer / 86400L;
5160026     remainder = *timer % 86400L;
5160027     //
5160028     // Minutes, after full days.
5160029     //
5160030     tms.tm_min = remainder / 60U;
5160031     //
5160032     // Seconds, after full minutes.
5160033     //
5160034     tms.tm_sec = remainder % 60U;
5160035     //
5160036     // Hours, after full days.
5160037     //
5160038     tms.tm_hour = tms.tm_min / 60;
5160039     //
5160040     // Minutes, after full hours.
5160041     //
5160042     tms.tm_min = tms.tm_min % 60;
5160043     //
5160044     // Find the week day. Must remove some days to align
5160045     // the

```

```

510046 // calculation. So: the week days of the first week
510047 // of 1970
510048 // are not valid! After 1970-01-04 calculations are
510049 // right.
510050 //
510051 tms.tm_wday = (days - 3) % 7;
510052 //
510053 // Find the year: the field 'tm_yday' will be
510054 // reduced to the days
510055 // of current year.
510056 //
510057 for (tms.tm_year = 1970; days > 0; tms.tm_year++)
510058 {
510059     if (leap_year (tms.tm_year))
510060     {
510061         if (days >= 366)
510062         {
510063             days -= 366;
510064             continue;
510065         }
510066         else
510067         {
510068             break;
510069         }
510070     }
510071     else
510072     {
510073         if (days >= 365)
510074         {
510075             days -= 365;
510076             continue;
510077         }
510078         else
510079         {
510080             break;
510081         }
510082     }
510083 }
510084 //
510085 // Day of the year.
510086 //
510087 tms.tm_yday = days + 1;
510088 //
510089 // Find the month.
510090 //
510091 tms.tm_mday = days + 1;
510092 //
510093 for (tms.tm_mon = 0, loop = 1; tms.tm_mon <= 12 && loop;)
510094 {
510095     tms.tm_mon++;
510096     //
510097     switch (tms.tm_mon)
510098     {
510099         case 1:
510100         case 3:
510101         case 5:
510102         case 7:
510103         case 8:
510104         case 10:
510105         case 12:
510106             if (tms.tm_mday >= 31)
510107             {
510108                 tms.tm_mday -= 31;
510109             }
510110             else
510111             {
510112                 loop = 0;
510113             }
510114             break;
510115         case 4:
510116         case 6:
510117         case 9:
510118         case 11:
510119             if (tms.tm_mday >= 30)
510120             {
510121                 tms.tm_mday -= 30;
510122             }
510123             else
510124             {
510125                 loop = 0;
510126             }
510127             break;
510128         case 2:
510129             if (leap_year (tms.tm_year))
510130             {
510131                 if (tms.tm_mday >= 29)
510132                 {

```

```

510033         tms.tm_mday -= 29;
510034     }
510035     else
510036     {
510037         loop = 0;
510038     }
510039 }
510040 else
510041 {
510042     if (tms.tm_mday >= 28)
510043     {
510044         tms.tm_mday -= 28;
510045     }
510046     else
510047     {
510048         loop = 0;
510049     }
510050 }
510051 break;
510052 }
510053 }
510054 //
510055 // No check for day light saving time.
510056 //
510057 tms.tm_isdst = 0;
510058 //
510059 // Return.
510060 //
510061 return (&tms);
510062 }
510063 }
510064 //-----
510065 static int
510066 leap_year (int year)
510067 {
510068     if ((year & 4) == 0)
510069     {
510070         if ((year & 100) == 0)
510071         {
510072             if ((year & 400) == 0)
510073             {
510074                 return (1);
510075             }
510076             else
510077             {
510078                 return (0);
510079             }
510080         }
510081         else
510082         {
510083             return (1);
510084         }
510085     }
510086     else
510087     {
510088         return (0);
510089     }
510090 }

```

## 95.29.4 lib/time/mktime.c

Si veda la sezione 88.15.

```

517001 #include <time.h>
517002 #include <string.h>
517003 #include <stdio.h>
517004 //-----
517005 static int leap_year (int year);
517006 //-----
517007 time_t
517008 mktime (const struct tm *timeptr)
517009 {
517010     time_t timer_total;
517011     time_t timer_aux;
517012     int days;
517013     int month;
517014     int year;
517015     //
517016     // From seconds to days.
517017     //
517018     timer_total = timeptr->tm_sec;
517019     //
517020     timer_aux = timeptr->tm_min;
517021     timer_aux *= 60;
517022     timer_total += timer_aux;
517023     //
517024     timer_aux = timeptr->tm_hour;

```

```

5170025 timer_aux *= (60 * 60);
5170026 timer_total += timer_aux;
5170027 //
5170028 timer_aux = timeptr->tm_mday;
5170029 timer_aux *= 24;
5170030 timer_aux *= (60 * 60);
5170031 timer_total += timer_aux;
5170032 //
5170033 // Month: add the days of months.
5170034 // Will scan the months, from the first, but before
5170035 // the
5170036 // months of the value inside field 'tm_mon'.
5170037 //
5170038 for (month = 1, days = 0; month < timeptr->tm_mon;
5170039 month++)
5170040 {
5170041     switch (month)
5170042     {
5170043     case 1:
5170044     case 3:
5170045     case 5:
5170046     case 7:
5170047     case 8:
5170048     case 10:
5170049         //
5170050         // There is no December, because the scan
5170051         // can go up to
5170052         // the month before the value inside field
5170053         // 'tm_mon'.
5170054         //
5170055         days += 31;
5170056         break;
5170057     case 4:
5170058     case 6:
5170059     case 9:
5170060     case 11:
5170061         days += 30;
5170062         break;
5170063     case 2:
5170064         if (leap_year (timeptr->tm_year))
5170065             {
5170066                 days += 29;
5170067             }
5170068         else
5170069             {
5170070                 days += 28;
5170071             }
5170072         break;
5170073     }
5170074 }
5170075 //
5170076 timer_aux = days;
5170077 timer_aux *= 24;
5170078 timer_aux *= (60 * 60);
5170079 timer_total += timer_aux;
5170080 //
5170081 // Year. The work is similar to the one of months:
5170082 // days of
5170083 // years are counted, up to the year before the one
5170084 // reported
5170085 // by the field 'tm_year'.
5170086 //
5170087 for (year = 1970, days = 0; year < timeptr->tm_year;
5170088 year++)
5170089 {
5170090     if (leap_year (year))
5170091     {
5170092         days += 366;
5170093     }
5170094     else
5170095     {
5170096         days += 365;
5170097     }
5170098 }
5170099 //
5170100 // After all, must subtract a day from the total.
5170101 //
5170102 days--;
5170103 //
5170104 timer_aux = days;
5170105 timer_aux *= 24;
5170106 timer_aux *= (60 * 60);
5170107 timer_total += timer_aux;
5170108 //
5170109 // That's all.
5170110 //
5170111 return (timer_total);

```

```

5170112 }
5170113 //-----
5170114 //-----
5170115 int
5170116 leap_year (int year)
5170117 {
5170118     if ((year % 4) == 0)
5170119     {
5170120         if ((year % 100) == 0)
5170121         {
5170122             if ((year % 400) == 0)
5170123             {
5170124                 return (1);
5170125             }
5170126             else
5170127             {
5170128                 return (0);
5170129             }
5170130         }
5170131         else
5170132         {
5170133             return (1);
5170134         }
5170135     }
5170136     else
5170137     {
5170138         return (0);
5170139     }
5170140 }

```

## 95.29.5 lib/time/stime.c

Si veda la sezione 87.59.

```

5180001 #include <time.h>
5180002 #include <sys/os32.h>
5180003 #include <errno.h>
5180004 //-----
5180005 int
5180006 stime (time_t * timer)
5180007 {
5180008     sysmsg_stime_t msg;
5180009     //
5180010     if (timer == NULL)
5180011     {
5180012         errset (EINVAL);
5180013         return (-1);
5180014     }
5180015     //
5180016     msg.timer = *timer;
5180017     msg.ret = 0;
5180018     sys (SYS_STIME, &msg, (sizeof msg));
5180019     return (msg.ret);
5180020 }

```

## 95.29.6 lib/time/time.c

Si veda la sezione 87.59.

```

5190001 #include <time.h>
5190002 #include <sys/os32.h>
5190003 //-----
5190004 time_t
5190005 time (time_t * timer)
5190006 {
5190007     sysmsg_time_t msg;
5190008     msg.ret = ((time_t) 0);
5190009     sys (SYS_TIME, &msg, (sizeof msg));
5190010     if (timer != NULL)
5190011     {
5190012         *timer = msg.ret;
5190013     }
5190014     return (msg.ret);
5190015 }

```

## 95.30 os32: «lib/unistd.h»

Si veda la sezione 91.3.

```

5200001 #ifndef _UNISTD_H
5200002 #define _UNISTD_H    1
5200003 //-----
5200004 #include <sys/stat.h>
5200005 #include <sys/types.h> // size_t, ssize_t, uid_t,
5200006 // gid_t, off_t, pid_t
5200007 #include <inttypes.h> // intptr_t

```

```

520008 #include <SEEK.h> // SEEK_CUR, SEEK_SET,
520009 // SEEK_END
520010
520011 //-----
520011 typedef unsigned int useconds_t; // This type
520012 // should be
520013 // used for
520014 // the
520015 // obsolete function
520016 // 'usleep()', that
520017 // is only
520018 // implemented inside
520019 // the
520020 // kernel, as
520021 // 'k_usleep()', for
520022 // the
520023 // drivers
520024 // management.
520025 //-----
520026 extern char **environ; // Variable 'environ' is used
520027 // by functions like
520028 // 'execv()' in replacement
520029 // for 'envp[][]'.
520030 //-----
520031 extern char *optarg; // Used by 'optarg()'.
520032 extern int optind; //
520033 extern int opterr; //
520034 extern int optopt; //
520035 //-----
520036 #define STDIN_FILENO 0 //
520037 #define STDOUT_FILENO 1 // Standard file
520038 // descriptors.
520039 #define STDERR_FILENO 2 //
520040 //-----
520041 #define R_OK 4 // Read permission.
520042 #define W_OK 2 // Write permission.
520043 #define X_OK 1 // Execute or traverse
520044 // permission.
520045 #define F_OK 0 // File exists.
520046 //-----
520047
520048 int access (const char *path, int mode);
520049 int brk (void *address);
520050 int chdir (const char *path);
520051 int chown (const char *path, uid_t uid, gid_t gid);
520052 int close (int fdn);
520053 int dup (int fdn_old);
520054 int dup2 (int fdn_old, int fdn_new);
520055 int execl (const char *path, char *arg, ...);
520056 int execlp (const char *path, char *arg, ...);
520057 int execlp (const char *path, char *arg, ...);
520058 int execv (const char *path, char *const argv[]);
520059 int execve (const char *path, char *const argv[],
520060 char *const envp[]);
520061 int execvp (const char *path, char *const argv[]);
520062 void _exit (int status);
520063 int fchown (int fdn, uid_t uid, gid_t gid);
520064 pid_t fork (void);
520065 char *getcwd (char *buffer, size_t size);
520066 gid_t getegid (void);
520067 uid_t geteuid (void);
520068 gid_t getgid (void);
520069 int getopt (int argc, char *const argv[],
520070 const char *optstring);
520071 pid_t getpgrp (void);
520072 pid_t getppid (void);
520073 pid_t getpid (void);
520074 uid_t getuid (void);
520075 int isatty (int fdn);
520076 int link (const char *path_old, const char *path_new);
520077 off_t lseek (int fdn, off_t offset, int whence);
520078 #define nice(n) (0)
520079 int pipe (int pipefd[2]);
520080 ssize_t read (int fdn, void *buffer, size_t count);
520081 #define readlink(p,b,s) ((ssize_t) -1)
520082 int rmdir (const char *path);
520083 void *sbrk (intptr_t increment);
520084 int setegid (gid_t gid);
520085 int seteuid (uid_t uid);
520086 int setgid (gid_t gid);
520087 int setpgrp (void);
520088 int setuid (uid_t uid);
520089 unsigned int sleep (unsigned int s);
520090 #define sync() /**/
520091 char *ttyname (int fdn);
520092 int unlink (const char *path);
520093 ssize_t write (int fdn, const void *buffer, size_t count);
520094 //-----

```

520095	#endif		
95.30.1	lib/unistd/_exit.c	.....	915
95.30.2	lib/unistd/access.c	.....	916
95.30.3	lib/unistd/brk.c	.....	916
95.30.4	lib/unistd/chdir.c	.....	917
95.30.5	lib/unistd/chown.c	.....	917
95.30.6	lib/unistd/close.c	.....	917
95.30.7	lib/unistd/dup.c	.....	918
95.30.8	lib/unistd/dup2.c	.....	918
95.30.9	lib/unistd/envIRON.c	.....	918
95.30.10	lib/unistd/execle.c	.....	918
95.30.11	lib/unistd/execl.c	.....	919
95.30.12	lib/unistd/execlp.c	.....	919
95.30.13	lib/unistd/execv.c	.....	920
95.30.14	lib/unistd/execve.c	.....	920
95.30.15	lib/unistd/execvp.c	.....	921
95.30.16	lib/unistd/fchdir.c	.....	922
95.30.17	lib/unistd/fchown.c	.....	922
95.30.18	lib/unistd/fork.c	.....	922
95.30.19	lib/unistd/getcwd.c	.....	923
95.30.20	lib/unistd/getegid.c	.....	923
95.30.21	lib/unistd/geteuid.c	.....	924
95.30.22	lib/unistd/getgid.c	.....	924
95.30.23	lib/unistd/getopt.c	.....	924
95.30.24	lib/unistd/getpgrp.c	.....	927
95.30.25	lib/unistd/getpid.c	.....	927
95.30.26	lib/unistd/getppid.c	.....	928
95.30.27	lib/unistd/getuid.c	.....	928
95.30.28	lib/unistd/isatty.c	.....	928
95.30.29	lib/unistd/link.c	.....	929
95.30.30	lib/unistd/lseek.c	.....	929
95.30.31	lib/unistd/pipe.c	.....	929
95.30.32	lib/unistd/read.c	.....	930
95.30.33	lib/unistd/rmdir.c	.....	931
95.30.34	lib/unistd/sbrk.c	.....	931
95.30.35	lib/unistd/setegid.c	.....	932
95.30.36	lib/unistd/seteuid.c	.....	932
95.30.37	lib/unistd/setgid.c	.....	932
95.30.38	lib/unistd/setpgrp.c	.....	932
95.30.39	lib/unistd/setuid.c	.....	933
95.30.40	lib/unistd/sleep.c	.....	933
95.30.41	lib/unistd/ttyname.c	.....	933
95.30.42	lib/unistd/unlink.c	.....	934
95.30.43	lib/unistd/write.c	.....	934

95.30.1 lib/unistd/\_exit.c

Si veda la sezione 87.2.

```

521001 #include <unistd.h>
521002 #include <sys/os32.h>
521003 //-----
521004 void
521005 _exit (int status)
521006 {
521007     sysmsg_exit_t msg;

```

```

521008 //
521009 // Only the low eight bit are returned.
521010 //
521011 msg.status = (status & 0xFF);
521012 //
521013 //
521014 //
521015 sys (SYS_EXIT, &msg, (sizeof msg));
521016 //
521017 // Should not return from system call, but if it
521018 // does, loop
521019 // forever:
521020 //
521021 while (1);
521022 }

```

### 95.30.2 lib/unistd/access.c

Si veda la sezione 88.4.

```

522001 #include <unistd.h>
522002 #include <sys/stat.h>
522003 #include <errno.h>
522004 -----
522005 int
522006 access (const char *path, int mode)
522007 {
522008     struct stat st;
522009     int status;
522010     uid_t euid;
522011 //
522012 status = stat (path, &st);
522013 if (status != 0)
522014     {
522015         return (-1);
522016     }
522017 //
522018 // File exists?
522019 //
522020 if (mode == F_OK)
522021     {
522022         return (0);
522023     }
522024 //
522025 // Some access permissions are requested: get
522026 // effective user id.
522027 //
522028 euid = geteuid ();
522029 //
522030 // Check owner access permissions.
522031 //
522032 if (st.st_uid == euid
522033     && ((st.st_mode & S_IRWXU) == (mode << 6)))
522034     {
522035         return (0);
522036     }
522037 //
522038 // Check others access permissions.
522039 //
522040 if ((st.st_mode & S_IRWXO) == (mode))
522041     {
522042         return (0);
522043     }
522044 //
522045 // Otherwise there are no access permissions.
522046 //
522047 errset (EACCES); // Permission denied.
522048 return (-1);
522049 }

```

### 95.30.3 lib/unistd/brk.c

Si veda la sezione 87.5.

```

523001 #include <unistd.h>
523002 #include <string.h>
523003 #include <sys/os32.h>
523004 #include <errno.h>
523005 #include <limits.h>
523006 -----
523007 int
523008 brk (void *address)
523009 {
523010     sysmsg_brk_t msg;
523011 //
523012 if (address == NULL)
523013     {

```

```

523014         errset (EINVAL);
523015         return (-1);
523016     }
523017 //
523018 msg.address = address;
523019 //
523020 sys (SYS_BRK, &msg, (sizeof msg));
523021 //
523022 errno = msg.errno;
523023 errln = msg.errln;
523024 strncpy (errfn, msg.errfn, PATH_MAX);
523025 return (msg.ret);
523026 }

```

### 95.30.4 lib/unistd/chdir.c

Si veda la sezione 87.6.

```

524001 #include <unistd.h>
524002 #include <string.h>
524003 #include <sys/os32.h>
524004 #include <errno.h>
524005 #include <limits.h>
524006 -----
524007 int
524008 chdir (const char *path)
524009 {
524010     sysmsg_chdir_t msg;
524011 //
524012 msg.path = path;
524013 msg.ret = 0;
524014 msg.errno = 0;
524015 //
524016 sys (SYS_CHDIR, &msg, (sizeof msg));
524017 //
524018 errno = msg.errno;
524019 errln = msg.errln;
524020 strncpy (errfn, msg.errfn, PATH_MAX);
524021 return (msg.ret);
524022 }

```

### 95.30.5 lib/unistd/chown.c

Si veda la sezione 87.8.

```

525001 #include <unistd.h>
525002 #include <string.h>
525003 #include <sys/os32.h>
525004 #include <errno.h>
525005 #include <limits.h>
525006 -----
525007 int
525008 chown (const char *path, uid_t uid, gid_t gid)
525009 {
525010     sysmsg_chown_t msg;
525011 //
525012 msg.path = path;
525013 msg.uid = uid;
525014 msg.gid = gid;
525015 //
525016 sys (SYS_CHOWN, &msg, (sizeof msg));
525017 //
525018 errno = msg.errno;
525019 errln = msg.errln;
525020 strncpy (errfn, msg.errfn, PATH_MAX);
525021 return (msg.ret);
525022 }

```

### 95.30.6 lib/unistd/close.c

Si veda la sezione 87.10.

```

526001 #include <unistd.h>
526002 #include <errno.h>
526003 #include <sys/os32.h>
526004 #include <string.h>
526005 -----
526006 int
526007 close (int fdn)
526008 {
526009     sysmsg_close_t msg;
526010     msg.fdn = fdn;
526011 //
526012 while (1)
526013     {
526014         sys (SYS_CLOSE, &msg, (sizeof msg));
526015         if (msg.ret < 0 && (msg.errno == EINPROGRESS

```

```

5260016         || msg.errno == EALREADY))
5260017     {
5260018         continue;
5260019     }
5260020     //
5260021     break;
5260022 }
5260023 errno = msg.errno;
5260024 errln = msg.errln;
5260025 strncpy (errfn, msg.errfn, PATH_MAX);
5260026 return (msg.ret);
5260027 }

```

### 95.30.7 lib/unistd/dup.c

&lt;&lt;

Si veda la sezione 87.12.

```

5270001 #include <unistd.h>
5270002 #include <sys/os32.h>
5270003 #include <string.h>
5270004 #include <errno.h>
5270005 //-----
5270006 int
5270007 dup (int fdn_old)
5270008 {
5270009     sysmsg_dup_t msg;
5270010     //
5270011     msg.fdn_old = fdn_old;
5270012     //
5270013     sys (SYS_DUP, &msg, (sizeof msg));
5270014     //
5270015     errno = msg.errno;
5270016     errln = msg.errln;
5270017     strncpy (errfn, msg.errfn, PATH_MAX);
5270018     return (msg.ret);
5270019 }

```

### 95.30.8 lib/unistd/dup2.c

&lt;&lt;

Si veda la sezione 87.12.

```

5280001 #include <unistd.h>
5280002 #include <sys/os32.h>
5280003 #include <string.h>
5280004 #include <errno.h>
5280005 //-----
5280006 int
5280007 dup2 (int fdn_old, int fdn_new)
5280008 {
5280009     sysmsg_dup2_t msg;
5280010     //
5280011     msg.fdn_old = fdn_old;
5280012     msg.fdn_new = fdn_new;
5280013     //
5280014     sys (SYS_DUP2, &msg, (sizeof msg));
5280015     //
5280016     errno = msg.errno;
5280017     errln = msg.errln;
5280018     strncpy (errfn, msg.errfn, PATH_MAX);
5280019     return (msg.ret);
5280020 }

```

### 95.30.9 lib/unistd/environ.c

&lt;&lt;

Si veda la sezione 91.1.

```

5290001 #include <unistd.h>
5290002 //-----
5290003 char **environ;

```

### 95.30.10 lib/unistd/execl.c

&lt;&lt;

Si veda la sezione 88.21.

```

5300001 #include <unistd.h>
5300002 #include <limits.h>
5300003 #include <stdarg.h>
5300004 #include <stddef.h>
5300005 //-----
5300006 int
5300007 execl (const char *path, char *arg, ...)
5300008 {
5300009     int argc;
5300010     char *arg_next;
5300011     char *argv[ARG_MAX / 2];
5300012     //

```

```

5300013     va_list ap;
5300014     va_start (ap, arg);
5300015     //
5300016     arg_next = arg;
5300017     //
5300018     for (argc = 0; argc < ARG_MAX / 2; argc++)
5300019     {
5300020         argv[argc] = arg_next;
5300021         if (argv[argc] == NULL)
5300022         {
5300023             break;           // End of arguments.
5300024         }
5300025         arg_next = va_arg (ap, char *);
5300026     }
5300027     //
5300028     return (execve (path, argv, environ)); // [1]
5300029 }
5300030 //
5300031 //
5300032 // The variable 'environ' is declared as
5300033 // 'char **environ' and is
5300034 // included from <unistd.h>.
5300035 //

```

### 95.30.11 lib/unistd/execl.c

&lt;&lt;

Si veda la sezione 88.21.

```

5310001 #include <unistd.h>
5310002 #include <limits.h>
5310003 #include <stdarg.h>
5310004 #include <stddef.h>
5310005 //-----
5310006 int
5310007 execl (const char *path, char *arg, ...)
5310008 {
5310009     int argc;
5310010     char *arg_next;
5310011     char *argv[ARG_MAX / 2];
5310012     char **envp;
5310013     //
5310014     va_list ap;
5310015     va_start (ap, arg);
5310016     //
5310017     arg_next = arg;
5310018     //
5310019     for (argc = 0; argc < ARG_MAX / 2; argc++)
5310020     {
5310021         argv[argc] = arg_next;
5310022         if (argv[argc] == NULL)
5310023         {
5310024             break;           // End of arguments.
5310025         }
5310026         arg_next = va_arg (ap, char *);
5310027     }
5310028     //
5310029     envp = va_arg (ap, char **);
5310030     //
5310031     return (execve (path, argv, envp));
5310032 }

```

### 95.30.12 lib/unistd/execlp.c

&lt;&lt;

Si veda la sezione 88.21.

```

5320001 #include <unistd.h>
5320002 #include <string.h>
5320003 #include <stdlib.h>
5320004 #include <errno.h>
5320005 #include <sys/os32.h>
5320006 //-----
5320007 int
5320008 execlp (const char *path, char *arg, ...)
5320009 {
5320010     int argc;
5320011     char *arg_next;
5320012     char *argv[ARG_MAX / 2];
5320013     char command[PATH_MAX];
5320014     int status;
5320015     //
5320016     va_list ap;
5320017     va_start (ap, arg);
5320018     //
5320019     arg_next = arg;
5320020     //
5320021     for (argc = 0; argc < ARG_MAX / 2; argc++)
5320022     {

```

```

532023     argv[argc] = arg_next;
532024     if (argv[argc] == NULL)
532025     {
532026         break;           // End of arguments.
532027     }
532028     arg_next = va_arg (ap, char *);
532029 }
532030 //
532031 // Get a full command path if necessary.
532032 //
532033 status = namep (path, command, (size_t) PATH_MAX);
532034 if (status != 0)
532035 {
532036     //
532037     // Variable 'errno' is already set by
532038     // 'commandp()'.
532039     //
532040     return (-1);
532041 }
532042 //
532043 // Return calling 'execve()'
532044 //
532045 return (execve (command, argv, environ)); // [1]
532046 }
532047 //
532048 // The variable 'environ' is declared as
532049 // 'char **environ' and is
532050 // included from <unistd.h>.
532051 //
532052 //

```

## 95.30.13 lib/unistd/execvc

&lt;&lt;

Si veda la sezione 88.21.

```

533001 #include <unistd.h>
533002 //-----
533003 int
533004 execv (const char *path, char *const argv[])
533005 {
533006     return (execve (path, argv, environ)); // [1]
533007 }
533008 //
533009 // The variable 'environ' is declared as
533010 // 'char **environ' and is
533011 // included from <unistd.h>.
533012 //
533013 //

```

## 95.30.14 lib/unistd/execve.c

&lt;&lt;

Si veda la sezione 87.14.

```

534001 #include <unistd.h>
534002 #include <sys/types.h>
534003 #include <sys/os32.h>
534004 #include <errno.h>
534005 #include <string.h>
534006 #include <string.h>
534007 //-----
534008 int
534009 execve (const char *path, char *const argv[],
534010        char *const envp[])
534011 {
534012     sysmsg_exec_t msg;
534013     size_t size;
534014     size_t arg_size;
534015     int argc;
534016     size_t env_size;
534017     int envc;
534018     char *arg_data = msg.arg_data;
534019     char *env_data = msg.env_data;
534020     //
534021     msg.path = path;
534022     msg.ret = 0;
534023     msg.errno = 0;
534024     //
534025     // Copy 'argv[]' inside a the message buffer
534026     // 'msg.arg_data',
534027     // separating each string with a null character and
534028     // counting the
534029     // number of strings inside 'argv'.
534030     //
534031     for (argc = 0, arg_size = 0, size = 0;
534032          argv != NULL &&
534033          argc < (ARG_MAX / 16) &&
534034          arg_size < ARG_MAX / 2 &&

```

```

534035     argv[argc] != NULL; argc++, arg_size += size)
534036     {
534037         size = strlen (argv[argc]);
534038         size++; // Count also the final null
534039         // character.
534040         if (size > (ARG_MAX / 2 - arg_size))
534041         {
534042             errset (E2BIG); // Argument list too
534043             // long.
534044             return (-1);
534045         }
534046         strncpy (arg_data, argv[argc], size);
534047         arg_data += size;
534048     }
534049     msg.argc = argc;
534050     //
534051     // Copy 'envp[]' inside a the message buffer
534052     // 'msg.env_data',
534053     // separating each string with a null character and
534054     // counting the
534055     // number of strings inside 'envc'.
534056     //
534057     for (envc = 0, env_size = 0, size = 0;
534058          envp != NULL &&
534059          envc < (ARG_MAX / 16) &&
534060          env_size < ARG_MAX / 2 &&
534061          envp[envc] != NULL; envc++, env_size += size)
534062     {
534063         size = strlen (envp[envc]);
534064         size++; // Count also the final null
534065         // character.
534066         if (size > (ARG_MAX / 2 - env_size))
534067         {
534068             errset (E2BIG); // Argument list too
534069             // long.
534070             return (-1);
534071         }
534072         strncpy (env_data, envp[envc], size);
534073         env_data += size;
534074     }
534075     msg.envc = envc;
534076     //
534077     // System call.
534078     //
534079     sys (SYS_EXEC, &msg, (sizeof msg));
534080     //
534081     // Should not return, but if it does, then there is
534082     // an error.
534083     //
534084     errno = msg.errno;
534085     errln = msg.errln;
534086     strncpy (errfn, msg.errfn, PATH_MAX);
534087     return (msg.ret);
534088 }

```

## 95.30.15 lib/unistd/execvp.c

&gt;&gt;

Si veda la sezione 88.21.

```

535001 #include <unistd.h>
535002 #include <string.h>
535003 #include <stdlib.h>
535004 #include <errno.h>
535005 #include <sys/os32.h>
535006 //-----
535007 int
535008 execvp (const char *path, char *const argv[])
535009 {
535010     char command[PATH_MAX];
535011     int status;
535012     //
535013     // Get a full command path if necessary.
535014     //
535015     status = namep (path, command, (size_t) PATH_MAX);
535016     if (status != 0)
535017     {
535018         //
535019         // Variable 'errno' is already set by 'namep()'.
535020         //
535021         return (-1);
535022     }
535023     //
535024     // Return calling 'execve()'
535025     //
535026     return (execve (command, argv, environ)); // [1]
535027 }
535028 //

```

```

5350029 //
5350030 // The variable 'environ' is declared as
5350031 // 'char **environ' and is
5350032 // included from <unistd.h>.
5350033 //

```

## 95.30.16 lib/unistd/fchdir.c

&lt;

Si veda la sezione 87.6.

```

5360001 #include <unistd.h>
5360002 #include <errno.h>
5360003 //-----
5360004 int
5360005 fchdir (int fdn)
5360006 {
5360007 //
5360008 // os32 requires to keep track of the path for the
5360009 // current working
5360010 // directory. The standard function 'fchdir()' is
5360011 // not applicable.
5360012 //
5360013 errset (E_NOT_IMPLEMENTED);
5360014 return (-1);
5360015 }

```

## 95.30.17 lib/unistd/fchown.c

&lt;

Si veda la sezione 87.8.

```

5370001 #include <unistd.h>
5370002 #include <string.h>
5370003 #include <sys/os32.h>
5370004 #include <errno.h>
5370005 #include <limits.h>
5370006 //-----
5370007 int
5370008 fchown (int fdn, uid_t uid, gid_t gid)
5370009 {
5370010     sysmsg_fchown_t msg;
5370011 //
5370012     msg.fdn = fdn;
5370013     msg.uid = uid;
5370014     msg.gid = gid;
5370015 //
5370016     sys (SYS_FCHOWN, &msg, (sizeof msg));
5370017 //
5370018     errno = msg.errno;
5370019     errln = msg.errln;
5370020     strncpy (errfn, msg.errfn, PATH_MAX);
5370021     return (msg.ret);
5370022 }

```

## 95.30.18 lib/unistd/fork.c

&lt;

Si veda la sezione 87.19.

```

5380001 #include <unistd.h>
5380002 #include <sys/types.h>
5380003 #include <sys/os32.h>
5380004 #include <errno.h>
5380005 #include <string.h>
5380006 //-----
5380007 pid_t
5380008 fork (void)
5380009 {
5380010     sysmsg_fork_t msg;
5380011 //
5380012     // Set the return value for the child process.
5380013 //
5380014     msg.ret = 0;
5380015 //
5380016     // Do the system call.
5380017 //
5380018     sys (SYS_FORK, &msg, (sizeof msg));
5380019 //
5380020     // If the system call has successfully generated a
5380021     // copy of
5380022     // the original process, the following code is
5380023     // executed from
5380024     // the parent and the child. But the child has the
5380025     // 'msg'
5380026     // structure untouched, while the parent has, at
5380027     // least, the
5380028     // pid number inside 'msg.ret'.
5380029     // If the system call fails, there is no child, and
5380030     // the

```

```

5380031 // parent finds the return value equal to -1, with
5380032 // an
5380033 // error number.
5380034 //
5380035     errno = msg.errno;
5380036     errln = msg.errln;
5380037     strncpy (errfn, msg.errfn, PATH_MAX);
5380038     return (msg.ret);
5380039 }

```

## 95.30.19 lib/unistd/getcwd.c

Si veda la sezione 87.21.

&lt;

```

5390001 #include <unistd.h>
5390002 #include <sys/types.h>
5390003 #include <sys/os32.h>
5390004 #include <errno.h>
5390005 #include <stddef.h>
5390006 #include <string.h>
5390007 //-----
5390008 char *
5390009 getcwd (char *buffer, size_t size)
5390010 {
5390011     sysmsg_uarea_t msg;
5390012 //
5390013     // Check arguments: the buffer must be given.
5390014 //
5390015     if (buffer == NULL)
5390016     {
5390017         errset (EINVAL);
5390018         return (NULL);
5390019     }
5390020 //
5390021     msg.path_cwd = buffer;
5390022     msg.path_cwd_size = size;
5390023 //
5390024     // Set the last buffer element to zero, for later
5390025     // verification.
5390026 //
5390027     buffer[size - 1] = 0;
5390028 //
5390029     // Just get the user area data.
5390030 //
5390031     sys (SYS_UAREA, &msg, (sizeof msg));
5390032 //
5390033     // Check that the path is still correctly
5390034     // terminated. If it isn't,
5390035     // the path is longer than the buffer size, because
5390036     // the last null
5390037     // character was overwritten.
5390038 //
5390039     if (buffer[size - 1] != 0)
5390040     {
5390041         errset (ERANGE);
5390042         return (NULL);
5390043     }
5390044 //
5390045     // Everything is fine.
5390046 //
5390047     return (buffer);
5390048 }

```

## 95.30.20 lib/unistd/getegid.c

&lt;

Si veda la sezione 87.22.

```

5400001 #include <unistd.h>
5400002 #include <sys/types.h>
5400003 #include <sys/os32.h>
5400004 #include <errno.h>
5400005 //-----
5400006 gid_t
5400007 getegid (void)
5400008 {
5400009     sysmsg_uarea_t msg;
5400010     msg.path_cwd = NULL;
5400011     msg.path_cwd_size = 0;
5400012     sys (SYS_UAREA, &msg, (sizeof msg));
5400013     return (msg.egid);
5400014 }

```



## 95.30.21 lib/unistd/geteuid.c

&lt;

Si veda la sezione 87.27.

```

5410001 #include <unistd.h>
5410002 #include <sys/types.h>
5410003 #include <sys/os32.h>
5410004 #include <errno.h>
5410005 //-----
5410006 uid_t
5410007 geteuid (void)
5410008 {
5410009     sysmsg_uarea_t msg;
5410010     msg.path_cwd = NULL;
5410011     msg.path_cwd_size = 0;
5410012     sys (SYS_UAREA, &msg, (sizeof msg));
5410013     return (msg.euid);
5410014 }

```

## 95.30.22 lib/unistd/getgid.c

&lt;

Si veda la sezione 87.22.

```

5420001 #include <unistd.h>
5420002 #include <sys/types.h>
5420003 #include <sys/os32.h>
5420004 #include <errno.h>
5420005 //-----
5420006 gid_t
5420007 getgid (void)
5420008 {
5420009     sysmsg_uarea_t msg;
5420010     msg.path_cwd = NULL;
5420011     msg.path_cwd_size = 0;
5420012     sys (SYS_UAREA, &msg, (sizeof msg));
5420013     return (msg.gid);
5420014 }

```

## 95.30.23 lib/unistd/getopt.c

&lt;

Si veda la sezione 88.56.

```

5430001 #include <unistd.h>
5430002 #include <sys/types.h>
5430003 #include <sys/os32.h>
5430004 #include <errno.h>
5430005 //-----
5430006 char *optarg;
5430007 int optind = 1;
5430008 int opterr = 1;
5430009 int optopt = 0;
5430010 //-----
5430011 static void getopt_no_argument (int opt);
5430012 //-----
5430013 int
5430014 getopt (int argc, char *const argv[], const char *optstring)
5430015 {
5430016     static int o = 0; // Index to scan grouped
5430017 // options.
5430018     int s; // Index to scan 'optstring'
5430019     int opt; // Current option letter.
5430020     int flag_argument; // If there should be an
5430021 // argument.
5430022 //
5430023 // Entering the function, 'flag_argument' is zero.
5430024 // Just to make
5430025 // it clear:
5430026 //
5430027 flag_argument = 0;
5430028 //
5430029 // Scan 'argv[]' elements, starting from the value
5430030 // that 'optind'
5430031 // already have.
5430032 //
5430033 for (; optind < argc; optind++)
5430034 {
5430035     //
5430036     // If an option is expected, some check must be
5430037     // done at
5430038     // the beginning.
5430039     //
5430040     if (!flag_argument)
5430041     {
5430042         //
5430043         // Check if the scan is finished and
5430044         // 'optind' should be kept
5430045         // untouched:
5430046         // 'argv[optind]' is a null pointer;

```

```

5430047 // 'argv[optind][0]' is not the character
5430048 // '-';
5430049 // 'argv[optind]' points to the string "--";
5430050 // all 'argv[]' elements are parsed.
5430051 //
5430052 if (argv[optind] == NULL
5430053     || argv[optind][0] != '-'
5430054     || argv[optind][1] == 0 || optind >= argc)
5430055     {
5430056         return (-1);
5430057     }
5430058 //
5430059 // Check if the scan is finished and
5430060 // 'optind' is to be
5430061 // incremented:
5430062 // 'argv[optind]' points to the string "--".
5430063 //
5430064 if (argv[optind][0] == '-'
5430065     && argv[optind][1] == '-'
5430066     && argv[optind][2] == 0)
5430067     {
5430068         optind++;
5430069         return (-1);
5430070     }
5430071 }
5430072 //
5430073 // Scan 'argv[optind]' using the static index
5430074 // 'o'.
5430075 //
5430076 for (; o < strlen (argv[optind]); o++)
5430077 {
5430078     //
5430079     // If there should be an option, index 'o'
5430080     // should
5430081     // start from 1, because 'argv[optind][0]'
5430082     // must
5430083     // be equal to '-'.
5430084     //
5430085     if (!flag_argument && (o == 0))
5430086     {
5430087         //
5430088         // As there is no options, 'o' cannot
5430089         // start
5430090         // from zero, so a new loop is done.
5430091         //
5430092         continue;
5430093     }
5430094     //
5430095     if (flag_argument)
5430096     {
5430097         //
5430098         // There should be an argument, starting
5430099         // from
5430100         // 'argv[optind][o]'.
5430101         //
5430102         if ((o == 0) && (argv[optind][o] == '-'))
5430103         {
5430104             //
5430105             // 'argv[optind][0]' is equal to
5430106             // '-', but there
5430107             // should be an argument instead:
5430108             // the argument
5430109             // is missing.
5430110             //
5430111             optarg = NULL;
5430112             //
5430113             if (optstring[0] == ':')
5430114             {
5430115                 //
5430116                 // As the option string starts
5430117                 // with ':' the
5430118                 // function must return ':'.
5430119                 //
5430120                 optopt = opt;
5430121                 opt = ':';
5430122             }
5430123             else
5430124             {
5430125                 //
5430126                 // As the option string does not
5430127                 // start with ':'
5430128                 // the function must return '?'.
5430129                 //
5430130                 getopt_no_argument (opt);
5430131                 optopt = opt;
5430132                 opt = '?';
5430133             }

```

```

5430134 //
5430135 // 'optind' is left untouched.
5430136 //
5430137 }
5430138 else
5430139 {
5430140 //
5430141 // The argument is found: 'optind'
5430142 // is to be
5430143 // incremented and 'o' is reset.
5430144 //
5430145 optarg = &argv[optind][o];
5430146 optind++;
5430147 o = 0;
5430148 }
5430149 //
5430150 // Return the option, or ':', or '?'.
5430151 //
5430152 return (opt);
5430153 }
5430154 else
5430155 {
5430156 //
5430157 // It should be an option: 'optstring[]'
5430158 // must be
5430159 // scanned.
5430160 //
5430161 opt = argv[optind][o];
5430162 //
5430163 for (s = 0, optopt = 0;
5430164      s < strlen (optstring); s++)
5430165 {
5430166 //
5430167 // If 'optstring[0]' is equal to ':',
5430168 // index 's' must
5430169 // start at 1.
5430170 //
5430171 if ((s == 0) && (optstring[0] == ':'))
5430172 {
5430173     continue;
5430174 }
5430175 //
5430176 if (opt == optstring[s])
5430177 {
5430178     //
5430179     if (optstring[s + 1] == ':')
5430180     {
5430181         //
5430182         // There is an argument.
5430183         //
5430184         flag_argument = 1;
5430185         break;
5430186     }
5430187     else
5430188     {
5430189         //
5430190         // There is no argument.
5430191         //
5430192         o++;
5430193         return (opt);
5430194     }
5430195 }
5430196 }
5430197 //
5430198 if (s >= strlen (optstring))
5430199 {
5430200 //
5430201 // The 'optstring' scan is concluded
5430202 // with no
5430203 // match.
5430204 //
5430205 o++;
5430206 optopt = opt;
5430207 return ('?');
5430208 }
5430209 //
5430210 // Otherwise the loop was broken.
5430211 //
5430212 }
5430213 }
5430214 //
5430215 // Check index 'o'.
5430216 //
5430217 if (o >= strlen (argv[optind]))
5430218 {
5430219     //
5430220     // There are no more options or there is no

```

```

5430221 // argument
5430222 // inside current 'argv[optind]' string.
5430223 // Index 'o' is
5430224 // reset before the next loop.
5430225 //
5430226 o = 0;
5430227 }
5430228 }
5430229 //
5430230 // No more elements inside 'argv' or loop broken:
5430231 // there might be a
5430232 // missing argument.
5430233 //
5430234 if (flag_argument)
5430235 {
5430236 //
5430237 // Missing option argument.
5430238 //
5430239 optarg = NULL;
5430240 //
5430241 if (optstring[0] == ':')
5430242 {
5430243     return (':');
5430244 }
5430245 else
5430246 {
5430247     getopt_no_argument (opt);
5430248     return ('?');
5430249 }
5430250 }
5430251 //
5430252 return (-1);
5430253 }
5430254 }
5430255 -----
5430256 static void
5430257 getopt_no_argument (int opt)
5430258 {
5430259     if (opterr)
5430260     {
5430261         fprintf (stderr,
5430262                 "Missing argument for option '-%c'\n", opt);
5430263     }
5430264 }

```

## 95.30.24 lib/unistd/getpgrp.c

Si veda la sezione 87.25. «

```

5440001 #include <unistd.h>
5440002 #include <sys/types.h>
5440003 #include <sys/os32.h>
5440004 #include <errno.h>
5440005 -----
5440006 pid_t
5440007 getpgrp (void)
5440008 {
5440009     sysmsg_uarea_t msg;
5440010     msg.path_cwd = NULL;
5440011     msg.path_cwd_size = 0;
5440012     sys (SYS_UAREA, &msg, (sizeof msg));
5440013     return (msg.pgrp);
5440014 }

```

## 95.30.25 lib/unistd/getpid.c

Si veda la sezione 87.25. «

```

5450001 #include <unistd.h>
5450002 #include <sys/types.h>
5450003 #include <sys/os32.h>
5450004 #include <errno.h>
5450005 -----
5450006 pid_t
5450007 getpid (void)
5450008 {
5450009     sysmsg_uarea_t msg;
5450010     msg.path_cwd = NULL;
5450011     msg.path_cwd_size = 0;
5450012     sys (SYS_UAREA, &msg, (sizeof msg));
5450013     return (msg.pid);
5450014 }

```

## 95.30.26 lib/unistd/getppid.c

« Si veda la sezione 87.25.

```

5460001 #include <unistd.h>
5460002 #include <sys/types.h>
5460003 #include <sys/os32.h>
5460004 #include <errno.h>
5460005 //-----
5460006 pid_t
5460007 getppid (void)
5460008 {
5460009     sysmsg_uarea_t msg;
5460010     msg.path_cwd = NULL;
5460011     msg.path_cwd_size = 0;
5460012     sys (SYS_UAREA, &msg, (sizeof msg));
5460013     return (msg.ppid);
5460014 }

```

## 95.30.27 lib/unistd/getuid.c

« Si veda la sezione 87.27.

```

5470001 #include <unistd.h>
5470002 #include <sys/types.h>
5470003 #include <sys/os32.h>
5470004 #include <errno.h>
5470005 //-----
5470006 uid_t
5470007 getuid (void)
5470008 {
5470009     sysmsg_uarea_t msg;
5470010     msg.path_cwd = NULL;
5470011     msg.path_cwd_size = 0;
5470012     sys (SYS_UAREA, &msg, (sizeof msg));
5470013     return (msg.uid);
5470014 }

```

## 95.30.28 lib/unistd/isatty.c

« Si veda la sezione 88.69.

```

5480001 #include <sys/stat.h>
5480002 #include <sys/os32.h>
5480003 #include <unistd.h>
5480004 #include <sys/types.h>
5480005 #include <errno.h>
5480006 //-----
5480007 int
5480008 isatty (int fdn)
5480009 {
5480010     struct stat file_status;
5480011     //
5480012     // Verify to have valid input data.
5480013     //
5480014     if (fdn < 0)
5480015     {
5480016         errset (EBADF);
5480017         return (0);
5480018     }
5480019     //
5480020     // Verify the standard input.
5480021     //
5480022     if (fstat (fdn, &file_status) == 0)
5480023     {
5480024         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
5480025         {
5480026             return (1); // Meaning it is ok!
5480027         }
5480028         if (major (file_status.st_rdev) == DEV_TTY_MAJOR)
5480029         {
5480030             return (1); // Meaning it is ok!
5480031         }
5480032     }
5480033     else
5480034     {
5480035         errset (errno);
5480036         return (0);
5480037     }
5480038     //
5480039     // If here, it is not a terminal of any kind.
5480040     //
5480041     errset (EINVAL);
5480042     return (0);
5480043 }

```

## 95.30.29 lib/unistd/link.c

« Si veda la sezione 87.30.

```

5490001 #include <unistd.h>
5490002 #include <string.h>
5490003 #include <sys/os32.h>
5490004 #include <errno.h>
5490005 #include <limits.h>
5490006 //-----
5490007 int
5490008 link (const char *path_old, const char *path_new)
5490009 {
5490010     sysmsg_link_t msg;
5490011     //
5490012     msg.path_old = path_old;
5490013     msg.path_new = path_new;
5490014     //
5490015     sys (SYS_LINK, &msg, (sizeof msg));
5490016     //
5490017     errno = msg.errno;
5490018     errln = msg.errln;
5490019     strncpy (errfn, msg.errfn, PATH_MAX);
5490020     return (msg.ret);
5490021 }

```

## 95.30.30 lib/unistd/lseek.c

« Si veda la sezione 87.33.

```

5500001 #include <unistd.h>
5500002 #include <sys/types.h>
5500003 #include <sys/os32.h>
5500004 #include <errno.h>
5500005 #include <string.h>
5500006 //-----
5500007 off_t
5500008 lseek (int fdn, off_t offset, int whence)
5500009 {
5500010     sysmsg_lseek_t msg;
5500011     msg.fdn = fdn;
5500012     msg.offset = offset;
5500013     msg.whence = whence;
5500014     sys (SYS_LSEEK, &msg, (sizeof msg));
5500015     errno = msg.errno;
5500016     errln = msg.errln;
5500017     strncpy (errfn, msg.errfn, PATH_MAX);
5500018     return (msg.ret);
5500019 }

```

## 95.30.31 lib/unistd/pipe.c

« Si veda la sezione 87.38.

```

5510001 #include <unistd.h>
5510002 #include <string.h>
5510003 #include <sys/os32.h>
5510004 #include <errno.h>
5510005 #include <limits.h>
5510006 //-----
5510007 int
5510008 pipe (int pipefd[2])
5510009 {
5510010     sysmsg_pipe_t msg;
5510011     //
5510012     if (pipefd == NULL)
5510013     {
5510014         errset (EINVAL);
5510015         return (-1);
5510016     }
5510017     //
5510018     sys (SYS_PIPE, &msg, (sizeof msg));
5510019     //
5510020     errno = msg.errno;
5510021     errln = msg.errln;
5510022     //
5510023     pipefd[0] = msg.pipefd[0];
5510024     pipefd[1] = msg.pipefd[1];
5510025     //
5510026     return (msg.ret);
5510027 }

```

« Si veda la sezione 87.39.

```

552001 #include <unistd.h>
552002 #include <sys/os32.h>
552003 #include <errno.h>
552004 #include <string.h>
552005 #include <stdio.h>
552006 #include <fcntl.h>
552007 -----
552008 ssize_t
552009 read (int fdn, void *buffer, size_t count)
552010 {
552011     sysmsg_read_t msg;
552012     //
552013     // Reduce size of read if necessary.
552014     //
552015     if (count > BUFSIZ)
552016     {
552017         count = BUFSIZ;
552018     }
552019     //
552020     // Fill the message.
552021     //
552022     msg.fdn = fdn;
552023     msg.buffer = buffer;
552024     msg.count = count;
552025     msg.fl_flags = 0; // Not necessary.
552026     msg.ret = 0;
552027     //
552028     // Repeat syscall, until something is received or
552029     // end of file is
552030     // reached.
552031     //
552032     while (1)
552033     {
552034         sys (SYS_READ, &msg, (sizeof msg));
552035         if (msg.ret == 0)
552036         {
552037             //
552038             // End of file.
552039             //
552040             break;
552041         }
552042         if (msg.ret < 0
552043             && (msg.errno == EAGAIN
552044                 || msg.errno == EWOULDBLOCK))
552045         {
552046             //
552047             // No data at the moment.
552048             //
552049             if (msg.fl_flags & O_NONBLOCK)
552050             {
552051                 //
552052                 // Don't block.
552053                 //
552054                 break;
552055             }
552056             else
552057             {
552058                 //
552059                 // Keep trying.
552060                 //
552061                 continue;
552062             }
552063         }
552064         //
552065         // Otherwise, we have read something.
552066         //
552067         break;
552068     }
552069     //
552070     //
552071     //
552072     if (msg.ret < 0)
552073     {
552074         //
552075         // No valid read.
552076         //
552077         errno = msg.errno;
552078         errln = msg.errln;
552079         strncpy (errfn, msg.errfn, PATH_MAX);
552080         return (msg.ret);
552081     }
552082     //
552083     if (msg.ret > count)
552084     {
552085         //

```

```

552086         // A strange value was returned. Considering it
552087         // a read error.
552088         //
552089         errset (EIO); // I/O error.
552090         return (-1);
552091     }
552092     //
552093     // A valid read: return.
552094     //
552095     return (msg.ret);
552096 }

```

## 95.30.33 lib/unistd/rmdir.c

« Si veda la sezione 87.41.

```

553001 #include <unistd.h>
553002 #include <string.h>
553003 #include <sys/os32.h>
553004 #include <errno.h>
553005 #include <limits.h>
553006 -----
553007 int
553008 rmdir (const char *path)
553009 {
553010     sysmsg_stat_t msg_stat;
553011     sysmsg_unlink_t msg_unlink;
553012     //
553013     msg_stat.path = path;
553014     //
553015     sys (SYS_STAT, &msg_stat, (sizeof msg_stat));
553016     //
553017     if (msg_stat.ret != 0)
553018     {
553019         errno = msg_stat.errno;
553020         errln = msg_stat.errln;
553021         strncpy (errfn, msg_stat.errfn, PATH_MAX);
553022         return (msg_stat.ret);
553023     }
553024     //
553025     if (!S_ISDIR (msg_stat.stat.st_mode))
553026     {
553027         errset (ENOTDIR); // Not a directory.
553028         return (-1);
553029     }
553030     //
553031     msg_unlink.path = path;
553032     //
553033     sys (SYS_UNLINK, &msg_unlink, (sizeof msg_unlink));
553034     //
553035     errno = msg_unlink.errno;
553036     errln = msg_unlink.errln;
553037     strncpy (errfn, msg_unlink.errfn, PATH_MAX);
553038     return (msg_unlink.ret);
553039 }

```

## 95.30.34 lib/unistd/sbrk.c

« Si veda la sezione 87.5.

```

554001 #include <unistd.h>
554002 #include <string.h>
554003 #include <sys/os32.h>
554004 #include <errno.h>
554005 #include <limits.h>
554006 -----
554007 void *
554008 sbrk (intptr_t increment)
554009 {
554010     sysmsg_sbrk_t msg_sbrk;
554011     //
554012     msg_sbrk.increment = increment;
554013     //
554014     sys (SYS_SBRK, &msg_sbrk, (sizeof msg_sbrk));
554015     //
554016     errno = msg_sbrk.errno;
554017     errln = msg_sbrk.errln;
554018     strncpy (errfn, msg_sbrk.errfn, PATH_MAX);
554019     return (msg_sbrk.ret);
554020 }

```

## 95.30.35 lib/unistd/setegid.c

« Si veda la sezione 87.48.

```

3550001 #include <unistd.h>
3550002 #include <sys/types.h>
3550003 #include <sys/os32.h>
3550004 #include <errno.h>
3550005 #include <string.h>
3550006 //-----
3550007 int
3550008 setegid (gid_t gid)
3550009 {
3550010     sysmsg_setegid_t msg;
3550011     msg.ret = 0;
3550012     msg.errno = 0;
3550013     msg.egid = gid;
3550014     sys (SYS_SETEGID, &msg, (sizeof msg));
3550015     errno = msg.errno;
3550016     errln = msg.errln;
3550017     strncpy (errfn, msg.errfn, PATH_MAX);
3550018     return (msg.ret);
3550019 }

```

## 95.30.36 lib/unistd/seteuid.c

« Si veda la sezione 87.51.

```

3560001 #include <unistd.h>
3560002 #include <sys/types.h>
3560003 #include <sys/os32.h>
3560004 #include <errno.h>
3560005 #include <string.h>
3560006 //-----
3560007 int
3560008 seteuid (uid_t uid)
3560009 {
3560010     sysmsg_seteuid_t msg;
3560011     msg.ret = 0;
3560012     msg.errno = 0;
3560013     msg.euid = uid;
3560014     sys (SYS_SETEUID, &msg, (sizeof msg));
3560015     errno = msg.errno;
3560016     errln = msg.errln;
3560017     strncpy (errfn, msg.errfn, PATH_MAX);
3560018     return (msg.ret);
3560019 }

```

## 95.30.37 lib/unistd/setgid.c

« Si veda la sezione 87.48.

```

3570001 #include <unistd.h>
3570002 #include <sys/types.h>
3570003 #include <sys/os32.h>
3570004 #include <errno.h>
3570005 #include <string.h>
3570006 //-----
3570007 int
3570008 setgid (gid_t gid)
3570009 {
3570010     sysmsg_setgid_t msg;
3570011     msg.ret = 0;
3570012     msg.errno = 0;
3570013     msg.egid = gid;
3570014     sys (SYS_SETGID, &msg, (sizeof msg));
3570015     errno = msg.errno;
3570016     errln = msg.errln;
3570017     strncpy (errfn, msg.errfn, PATH_MAX);
3570018     return (msg.ret);
3570019 }

```

## 95.30.38 lib/unistd/setpgrp.c

« Si veda la sezione 87.50.

```

3580001 #include <unistd.h>
3580002 #include <sys/os32.h>
3580003 #include <stddef.h>
3580004 //-----
3580005 int
3580006 setpgrp (void)
3580007 {
3580008     sys (SYS_PGRP, NULL, (size_t) 0);
3580009     return (0);
3580010 }

```

## 95.30.39 lib/unistd/setuid.c

« Si veda la sezione 87.51.

```

5590001 #include <unistd.h>
5590002 #include <sys/types.h>
5590003 #include <sys/os32.h>
5590004 #include <errno.h>
5590005 #include <string.h>
5590006 //-----
5590007 int
5590008 setuid (uid_t uid)
5590009 {
5590010     sysmsg_setuid_t msg;
5590011     msg.ret = 0;
5590012     msg.errno = 0;
5590013     msg.euid = uid;
5590014     sys (SYS_SETUID, &msg, (sizeof msg));
5590015     errno = msg.errno;
5590016     errln = msg.errln;
5590017     strncpy (errfn, msg.errfn, PATH_MAX);
5590018     return (msg.ret);
5590019 }

```

## 95.30.40 lib/unistd/sleep.c

« Si veda la sezione 87.53.

```

5600001 #include <unistd.h>
5600002 #include <sys/types.h>
5600003 #include <sys/os32.h>
5600004 #include <errno.h>
5600005 #include <time.h>
5600006 //-----
5600007 unsigned int
5600008 sleep (unsigned int seconds)
5600009 {
5600010     sysmsg_sleep_t msg;
5600011     time_t start;
5600012     time_t end;
5600013     int slept;
5600014     //
5600015     if (seconds == 0)
5600016     {
5600017         return (0);
5600018     }
5600019     //
5600020     msg.events = WAKEUP_EVENT_TIMER;
5600021     msg.seconds = seconds;
5600022     sys (SYS_SLEEP, &msg, (sizeof msg));
5600023     start = msg.ret;
5600024     end = time (NULL);
5600025     slept = end - msg.ret;
5600026     //
5600027     if (slept < 0)
5600028     {
5600029         return (seconds);
5600030     }
5600031     else if (slept < seconds)
5600032     {
5600033         return (seconds - slept);
5600034     }
5600035     else
5600036     {
5600037         return (0);
5600038     }
5600039 }

```

## 95.30.41 lib/unistd/ttyname.c

« Si veda la sezione 88.133.

```

5610001 #include <sys/os32.h>
5610002 #include <sys/stat.h>
5610003 #include <unistd.h>
5610004 #include <sys/types.h>
5610005 #include <errno.h>
5610006 #include <limits.h>
5610007 //-----
5610008 char *
5610009 ttyname (int fdn)
5610010 {
5610011     dev_t dev_minor;
5610012     struct stat file_status;
5610013     static char name[PATH_MAX];
5610014     //
5610015     // Verify to have valid input data.
5610016     //

```

```

5610017 if (fdn < 0)
5610018 {
5610019     errset (EBADF);
5610020     return (NULL);
5610021 }
5610022 //
5610023 // Verify the file descriptor.
5610024 //
5610025 if (fstat (fdn, &file_status) == 0)
5610026 {
5610027     if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
5610028     {
5610029         dev_minor = minor (file_status.st_rdev);
5610030         //
5610031         // If minor is equal to 0xFF, it is
5610032         // '/dev/console'.
5610033         //
5610034         if (dev_minor < 0xFF)
5610035         {
5610036             sprintf (name, "/dev/console%i", dev_minor);
5610037         }
5610038         else
5610039         {
5610040             strcpy (name, "/dev/console");
5610041         }
5610042         return (name);
5610043     }
5610044     else if (file_status.st_rdev == DEV_TTY)
5610045     {
5610046         strcpy (name, "/dev/tty");
5610047         return (name);
5610048     }
5610049     else
5610050     {
5610051         errset (ENOTTY);
5610052         return (NULL);
5610053     }
5610054 }
5610055 else
5610056 {
5610057     errset (errno);
5610058     return (NULL);
5610059 }
5610060 }

```

## 95.30.42 lib/unistd/unlink.c

Si veda la sezione 87.62.

```

5620001 #include <unistd.h>
5620002 #include <string.h>
5620003 #include <sys/os32.h>
5620004 #include <errno.h>
5620005 #include <limits.h>
5620006 //-----
5620007 int
5620008 unlink (const char *path)
5620009 {
5620010     sysmsg_unlink_t msg;
5620011     //
5620012     msg.path = path;
5620013     //
5620014     sys (SYS_UNLINK, &msg, (sizeof msg));
5620015     //
5620016     errno = msg.errno;
5620017     errln = msg.errln;
5620018     strncpy (errfn, msg.errfn, PATH_MAX);
5620019     return (msg.ret);
5620020 }

```

## 95.30.43 lib/unistd/write.c

Si veda la sezione 87.64.

```

5630001 #include <unistd.h>
5630002 #include <sys/os32.h>
5630003 #include <errno.h>
5630004 #include <string.h>
5630005 #include <stdio.h>
5630006 //-----
5630007 ssize_t
5630008 write (int fdn, const void *buffer, size_t count)
5630009 {
5630010     sysmsg_write_t msg;
5630011     //
5630012     // Reduce size of write if necessary.
5630013     //

```

```

5600014 if (count > BUFSIZ)
5600015 {
5600016     count = BUFSIZ;
5600017 }
5600018 //
5600019 // Fill the message.
5600020 //
5600021 msg.fdn = fdn;
5600022 msg.buffer = buffer;
5600023 msg.count = count;
5600024 //
5600025 // Syscall.
5600026 //
5600027 sys (SYS_WRITE, &msg, (sizeof msg));
5600028 //
5600029 // Check result and return.
5600030 //
5600031 if (msg.ret < 0)
5600032 {
5600033     //
5600034     // No valid write.
5600035     //
5600036     errno = msg.errno;
5600037     errln = msg.errln;
5600038     strncpy (errfn, msg.errfn, PATH_MAX);
5600039     return (msg.ret);
5600040 }
5600041 //
5600042 if (msg.ret > count)
5600043 {
5600044     //
5600045     // A strange value was returned. Considering it
5600046     // a read error.
5600047     //
5600048     errset (EIO); // I/O error.
5600049     return (-1);
5600050 }
5600051 //
5600052 // A valid write return.
5600053 //
5600054 return (msg.ret);
5600055 }

```

## 95.31 os32: «lib/utime.h»

Si veda la sezione 91.3.

```

5640001 #ifndef _UTIME_H
5640002 #define _UTIME_H 1
5640003 //-----
5640004 #include <restrict.h>
5640005 #include <sys/types.h> // time_t
5640006 //-----
5640007 struct utimbuf
5640008 {
5640009     time_t actime;
5640010     time_t modtime;
5640011 };
5640012 //-----
5640013 int utime (const char *path, const struct utimbuf *times);
5640014 //-----
5640015 #endif
5640016

```

## 95.31.1 lib/utime/utime.c ..... 935

## 95.31.1 lib/utime/utime.c

Si veda la sezione 91.3.

```

5650001 #include <utime.h>
5650002 #include <errno.h>
5650003 //-----
5650004 int
5650005 utime (const char *path, const struct utimbuf *times)
5650006 {
5650007     //
5650008     // Currently not implemented.
5650009     //
5650010     return (0);
5650011 }

```

## Sorgenti delle applicazioni

96.1	os32: directory «applic/»	938
96.1.1	applic/MAKEDEV.c	938
96.1.2	applic/aaa.c	940
96.1.3	applic/allocated.c	940
96.1.4	applic/arp.c	941
96.1.5	applic/bbb.c	942
96.1.6	applic/cat.c	942
96.1.7	applic/ccc.c	943
96.1.8	applic/chgrp.c	944
96.1.9	applic/chmod.c	945
96.1.10	applic/chown.c	946
96.1.11	applic/cp.c	947
96.1.12	applic/crt0.mer.s	949
96.1.13	applic/crt0.sep.s	951
96.1.14	applic/date.c	953
96.1.15	applic/ed.c	955
96.1.16	applic/getty.c	972
96.1.17	applic/http.c	973
96.1.18	applic/init.c	980
96.1.19	applic/ipconfig.c	983
96.1.20	applic/kill.c	984
96.1.21	applic/ln.c	987
96.1.22	applic/login.c	989
96.1.23	applic/ls.c	991
96.1.24	applic/man.c	995
96.1.25	applic/mkdir.c	999
96.1.26	applic/mmcheck.c	1002
96.1.27	applic/more.c	1004
96.1.28	applic/mount.c	1007
96.1.29	applic/nc.c	1008
96.1.30	applic/ping.c	1012
96.1.31	applic/ps.c	1015
96.1.32	applic/rm.c	1018
96.1.33	applic/rmdir.c	1019
96.1.34	applic/route.c	1020
96.1.35	applic/shell.c	1021
96.1.36	applic/t_fcntl.c	1024
96.1.37	applic/t_fifo.c	1024
96.1.38	applic/t_grp.c	1026
96.1.39	applic/t_nc.c	1026
96.1.40	applic/t_ping2.c	1030
96.1.41	applic/t_pipe.c	1031
96.1.42	applic/t_read.c	1032
96.1.43	applic/t_ret.c	1033
96.1.44	applic/t_rx_udp.c	1033
96.1.45	applic/t_scr.c	1034
96.1.46	applic/t_setjmp.c	1035
96.1.47	applic/t_sig.c	1035
96.1.48	applic/t_sig2.c	1036
96.1.49	applic/t_tx_tcp.c	1037
96.1.50	applic/t_tx_udp.c	1038
96.1.51	applic/touch.c	1039
96.1.52	applic/tty.c	1041
96.1.53	applic/umount.c	1041
96.1.54	applic/yes.c	1042

## 96.1 os32: directory «applic/»

&lt;&lt;

## 96.1.1 applic/MAKEDEV.c

&lt;

Si veda la sezione 92.6.

```

560001 #include <unistd.h>
560002 #include <stdlib.h>
560003 #include <sys/stat.h>
560004 #include <fcntl.h>
560005 #include <kernel/dev.h>
560006 #include <stdio.h>
560007 //-----
560008 int
560009 main (void)
560010 {
560011     int status;
560012     //
560013     status =
560014         mknod ("mem", (mode_t) (S_IFCHR | 0444),
560015             (dev_t) DEV_MEM);
560016     if (status)
560017         perror (NULL);
560018     status =
560019         mknod ("null", (mode_t) (S_IFCHR | 0666),
560020             (dev_t) DEV_NULL);
560021     if (status)
560022         perror (NULL);
560023     status =
560024         mknod ("port", (mode_t) (S_IFCHR | 0644),
560025             (dev_t) DEV_PORT);
560026     if (status)
560027         perror (NULL);
560028     status =
560029         mknod ("zero", (mode_t) (S_IFCHR | 0666),
560030             (dev_t) DEV_ZERO);
560031     if (status)
560032         perror (NULL);
560033     status =
560034         mknod ("tty", (mode_t) (S_IFCHR | 0666),
560035             (dev_t) DEV_TTY);
560036     if (status)
560037         perror (NULL);
560038     status = mknod ("kmem_ps", (mode_t) (S_IFCHR | 0444),
560039                 (dev_t) DEV_KMEM_PS);
560040     if (status)
560041         perror (NULL);
560042     status =
560043         mknod ("kmem_mmp", (mode_t) (S_IFCHR | 0444),
560044             (dev_t) DEV_KMEM_MMP);
560045     if (status)
560046         perror (NULL);
560047     status = mknod ("kmem_sb", (mode_t) (S_IFCHR | 0444),
560048                 (dev_t) DEV_KMEM_SB);
560049     if (status)
560050         perror (NULL);
560051     status =
560052         mknod ("kmem_inode", (mode_t) (S_IFCHR | 0444),
560053             (dev_t) DEV_KMEM_INODE);
560054     if (status)
560055         perror (NULL);
560056     status =
560057         mknod ("kmem_file", (mode_t) (S_IFCHR | 0444),
560058             (dev_t) DEV_KMEM_FILE);
560059     if (status)
560060         perror (NULL);
560061     status = mknod ("console", (mode_t) (S_IFCHR | 0644),
560062                 (dev_t) DEV_CONSOLE);
560063     if (status)
560064         perror (NULL);
560065     status =
560066         mknod ("console0", (mode_t) (S_IFCHR | 0644),
560067             (dev_t) DEV_CONSOLE0);
560068     if (status)
560069         perror (NULL);
560070     status =
560071         mknod ("console1", (mode_t) (S_IFCHR | 0644),
560072             (dev_t) DEV_CONSOLE1);
560073     if (status)
560074         perror (NULL);
560075     status =
560076         mknod ("console2", (mode_t) (S_IFCHR | 0644),
560077             (dev_t) DEV_CONSOLE2);
560078     if (status)
560079         perror (NULL);
560080     status =
560081         mknod ("console3", (mode_t) (S_IFCHR | 0644),
560082             (dev_t) DEV_CONSOLE3);

```

```

560083     if (status)
560084         perror (NULL);
560085
560086     status =
560087         mknod ("dm0", (mode_t) (S_IFBLK | 0644),
560088             (dev_t) DEV_DM0);
560089     if (status)
560090         perror (NULL);
560091     status =
560092         mknod ("dm01", (mode_t) (S_IFBLK | 0644),
560093             (dev_t) DEV_DM01);
560094     if (status)
560095         perror (NULL);
560096     status =
560097         mknod ("dm02", (mode_t) (S_IFBLK | 0644),
560098             (dev_t) DEV_DM02);
560099     if (status)
560100         perror (NULL);
560101     status =
560102         mknod ("dm03", (mode_t) (S_IFBLK | 0644),
560103             (dev_t) DEV_DM03);
560104     if (status)
560105         perror (NULL);
560106     status =
560107         mknod ("dm04", (mode_t) (S_IFBLK | 0644),
560108             (dev_t) DEV_DM04);
560109     if (status)
560110         perror (NULL);
560111     status =
560112         mknod ("dm10", (mode_t) (S_IFBLK | 0644),
560113             (dev_t) DEV_DM10);
560114     if (status)
560115         perror (NULL);
560116     status =
560117         mknod ("dm11", (mode_t) (S_IFBLK | 0644),
560118             (dev_t) DEV_DM11);
560119     if (status)
560120         perror (NULL);
560121     status =
560122         mknod ("dm12", (mode_t) (S_IFBLK | 0644),
560123             (dev_t) DEV_DM12);
560124     if (status)
560125         perror (NULL);
560126     status =
560127         mknod ("dm13", (mode_t) (S_IFBLK | 0644),
560128             (dev_t) DEV_DM13);
560129     if (status)
560130         perror (NULL);
560131     status =
560132         mknod ("dm14", (mode_t) (S_IFBLK | 0644),
560133             (dev_t) DEV_DM14);
560134     if (status)
560135         perror (NULL);
560136     status =
560137         mknod ("dm20", (mode_t) (S_IFBLK | 0644),
560138             (dev_t) DEV_DM20);
560139     if (status)
560140         perror (NULL);
560141     status =
560142         mknod ("dm21", (mode_t) (S_IFBLK | 0644),
560143             (dev_t) DEV_DM21);
560144     if (status)
560145         perror (NULL);
560146     status =
560147         mknod ("dm22", (mode_t) (S_IFBLK | 0644),
560148             (dev_t) DEV_DM22);
560149     if (status)
560150         perror (NULL);
560151     status =
560152         mknod ("dm23", (mode_t) (S_IFBLK | 0644),
560153             (dev_t) DEV_DM23);
560154     if (status)
560155         perror (NULL);
560156     status =
560157         mknod ("dm24", (mode_t) (S_IFBLK | 0644),
560158             (dev_t) DEV_DM24);
560159     if (status)
560160         perror (NULL);
560161     status =
560162         mknod ("dm30", (mode_t) (S_IFBLK | 0644),
560163             (dev_t) DEV_DM30);
560164     if (status)
560165         perror (NULL);
560166     status =
560167         mknod ("dm31", (mode_t) (S_IFBLK | 0644),
560168             (dev_t) DEV_DM31);
560169     if (status)

```



```

5600170     perror (NULL);
5600171     status =
5600172     mknod ("dm32", (mode_t) (S_IFBLK | 0644),
5600173           (dev_t) DEV_DM32);
5600174     if (status)
5600175         perror (NULL);
5600176     status =
5600177     mknod ("dm33", (mode_t) (S_IFBLK | 0644),
5600178           (dev_t) DEV_DM33);
5600179     if (status)
5600180         perror (NULL);
5600181     status =
5600182     mknod ("dm34", (mode_t) (S_IFBLK | 0644),
5600183           (dev_t) DEV_DM34);
5600184     if (status)
5600185         perror (NULL);
5600186     //
5600187     return (0);
5600188 }

```

## 96.1.2 applic/aaa.c

«

Si veda la sezione 86.1.

```

5670001 #include <unistd.h>
5670002 #include <stdio.h>
5670003 //-----
5670004 int
5670005 main (void)
5670006 {
5670007     unsigned int count;
5670008     for (count = 0; count < 60; count++)
5670009     {
5670010         printf ("a");
5670011         sleep (1);
5670012     }
5670013     return (8);
5670014 }

```

## 96.1.3 applic/allocated.c

«

Si veda la sezione 86.2.

```

5680001 #include <sys/os32.h>
5680002 #include <kernel/memory.h>
5680003 #include <unistd.h>
5680004 #include <stdio.h>
5680005 #include <fcntl.h>
5680006 #include <unistd.h>
5680007 #include <stdlib.h>
5680008 //-----
5680009 uint32_t mb_table[MEM_MAX_BLOCKS / 32]; // Memory
5680010                                         // blocks map.
5680011 unsigned int mb_max = MEM_MAX_BLOCKS; // Memory
5680012                                         // blocks max.
5680013 //-----
5680014 int
5680015 main (int argc, char *argv[], char *envp[])
5680016 {
5680017     unsigned int block;
5680018     unsigned int blocks = MEM_MAX_BLOCKS;
5680019     int i;
5680020     int j;
5680021     uint32_t mask;
5680022     unsigned int start = 0;
5680023     unsigned int stop = 0;
5680024     unsigned int status = 0;
5680025     int fd;
5680026     ssize_t size_read;
5680027     char *buffer = (char *) mb_table;
5680028     //
5680029     fd = open ("/dev/kmem_map", O_RDONLY);
5680030     if (fd < 0)
5680031     {
5680032         printf ("[%s] Cannot open \"/dev/kmem_map\" ",
5680033               argv[0]);
5680034         perror (NULL);
5680035         return (0);
5680036     }
5680037     //
5680038     lseek (fd, (off_t) 0, SEEK_SET);
5680039     for (i = 0; i < (MEM_MAX_BLOCKS / 8); i += size_read)
5680040     {
5680041         size_read = read (fd, &buffer[i], BUFSIZ);
5680042         if (size_read < 0)
5680043         {
5680044             printf

```

```

5680045         ("[%s] Cannot read "
5680046          "\"/dev/kmem_map\" %i %i ",
5680047          argv[0], size_read, sizeof (mb_table));
5680048         perror (NULL);
5680049         return (0);
5680050     }
5680051     }
5680052     //
5680053     printf ("Hex mem map, blocks of %x:", MEM_BLOCK_SIZE);
5680054     //
5680055     for (block = 0; block < blocks; block++)
5680056     {
5680057         i = block / 32;
5680058         j = block % 32;
5680059         mask = 0x80000000 >> j;
5680060         if (mb_table[i] & mask)
5680061         {
5680062             //
5680063             // Allocated block
5680064             //
5680065             if (status == 0)
5680066             {
5680067                 status = 1;
5680068                 start = block;
5680069             }
5680070         }
5680071         else
5680072         {
5680073             //
5680074             // Not allocated block.
5680075             //
5680076             if (status == 1)
5680077             {
5680078                 status = 0;
5680079                 stop = block;
5680080             }
5680081         }
5680082         //
5680083         //
5680084         //
5680085         if (stop > 0)
5680086         {
5680087             printf (" %x-%x", start, stop);
5680088             start = 0;
5680089             stop = 0;
5680090         }
5680091     }
5680092     printf ("\n");
5680093     //
5680094     return (0);
5680095 }

```

## 96.1.4 applic/arp.c

Si veda la sezione 92.1.

»

```

5690001 #include <sys/os32.h>
5690002 #include <kernel/net/arp.h>
5690003 #include <unistd.h>
5690004 #include <stdio.h>
5690005 #include <fcntl.h>
5690006 #include <unistd.h>
5690007 #include <stdlib.h>
5690008 #include <time.h>
5690009 //-----
5690010 int
5690011 main (int argc, char *argv[], char *envp[])
5690012 {
5690013     int fd;
5690014     ssize_t size_read;
5690015     char buffer[sizeof (arp_t)];
5690016     int a;
5690017     arp_t *arp_table_item;
5690018     //
5690019     // All options are ignored.
5690020     //
5690021     // Open '/dev/kmem_arp', to get the ARP table.
5690022     //
5690023     fd = open ("/dev/kmem_arp", O_RDONLY);
5690024     if (fd < 0)
5690025     {
5690026         printf ("[%s] Cannot open \"/dev/kmem_arp\" ",
5690027               argv[0]);
5690028         perror (NULL);
5690029         exit (0);
5690030     }
5690031     //

```

```

5690032 // Scan ARP items and then print body.
5690033 //
5690034 for (a = 0; a < ARP_MAX_ITEMS; a++)
5690035 {
5690036     lseek (fd, (off_t) a, SEEK_SET);
5690037     size_read = read (fd, buffer, sizeof (arp_t));
5690038     if (size_read < sizeof (arp_t))
5690039     {
5690040         printf
5690041             ("[%s] Cannot read "
5690042              "\"/dev/kmem_arp\" item %i ", argv[0], a);
5690043         perror (NULL);
5690044         continue;
5690045     }
5690046     arp_table_item = (arp_t *) buffer;
5690047     if (arp_table_item->time > 0)
5690048     {
5690049         printf ("%i.%i.%i.%i ",
5690050                 arp_table_item->ip >> 24 & 0x000000FF,
5690051                 arp_table_item->ip >> 16 & 0x000000FF,
5690052                 arp_table_item->ip >> 8 & 0x000000FF,
5690053                 arp_table_item->ip >> 0 & 0x000000FF);
5690054         //
5690055         printf ("%02x:%02x:%02x:%02x:%02x:%02x  ",
5690056                 arp_table_item->mac[0],
5690057                 arp_table_item->mac[1],
5690058                 arp_table_item->mac[2],
5690059                 arp_table_item->mac[3],
5690060                 arp_table_item->mac[4],
5690061                 arp_table_item->mac[5]);
5690062         //
5690063         printf ("%3us\n",
5690064                 (unsigned int) (time (NULL) -
5690065                                 arp_table_item->time));
5690066     }
5690067 }
5690068 close (fd);
5690069 return (0);
5690070 }

```

### 96.1.5 applic/bbb.c

&lt;&lt;

Si veda la sezione 86.1.

```

5700001 #include <unistd.h>
5700002 #include <stdio.h>
5700003 #include <stdlib.h>
5700004 //-----
5700005 int
5700006 main (void)
5700007 {
5700008     unsigned int count;
5700009     for (count = 0; count < 30; count++)
5700010     {
5700011         printf ("b");
5700012         sleep (2);
5700013     }
5700014     exit (0);
5700015     return (0);
5700016 }

```

### 96.1.6 applic/cat.c

&lt;&lt;

Si veda la sezione 86.4.

```

5710001 #include <fcntl.h>
5710002 #include <sys/stat.h>
5710003 #include <stddef.h>
5710004 #include <unistd.h>
5710005 #include <stdio.h>
5710006 #include <stdlib.h>
5710007 #include <errno.h>
5710008 //-----
5710009 static void cat_file_descriptor (int fd);
5710010 //-----
5710011 int
5710012 main (int argc, char *argv[], char *envp[])
5710013 {
5710014     int i;
5710015     int fd;
5710016     struct stat file_status;
5710017     //
5710018     // Check if the input comes from standard input.
5710019     //
5710020     if (argc < 2)
5710021     {
5710022         cat_file_descriptor (STDIN_FILENO);

```

```

5710023         exit (0);
5710024     }
5710025     //
5710026     // There is at least an argument: scan them.
5710027     //
5710028     for (i = 1; i < argc; i++)
5710029     {
5710030         //
5710031         // Verify if the file exists.
5710032         //
5710033         if (stat (argv[i], &file_status) != 0)
5710034         {
5710035             fprintf (stderr,
5710036                     "File \"%s\" does not exist!\n",
5710037                     argv[i]);
5710038             continue;
5710039         }
5710040         //
5710041         // File exists: check the file type.
5710042         //
5710043         if (S_ISDIR (file_status.st_mode))
5710044         {
5710045             fprintf (stderr, "Cannot \"cat\" "
5710046                     "\"%s\": it is a directory!\n", argv[i]);
5710047             continue;
5710048         }
5710049         //
5710050         // File exists and can be "cat"ed.
5710051         //
5710052         fd = open (argv[i], O_RDONLY);
5710053         if (fd >= 0)
5710054         {
5710055             cat_file_descriptor (fd);
5710056             close (fd);
5710057         }
5710058         else
5710059         {
5710060             perror (NULL);
5710061             exit (1);
5710062         }
5710063     }
5710064     return (0);
5710065 }
5710066 //-----
5710067 //-----
5710068 static void
5710069 cat_file_descriptor (int fd)
5710070 {
5710071     ssize_t count;
5710072     char buffer[BUFSIZ];
5710073
5710074     for (;;)
5710075     {
5710076         count = read (fd, buffer, (size_t) BUFSIZ);
5710077         if (count > 0)
5710078         {
5710079             write (STDOUT_FILENO, buffer, (size_t) count);
5710080         }
5710081         else
5710082         {
5710083             break;
5710084         }
5710085     }
5710086 }

```

### 96.1.7 applic/coc.c

&lt;&lt;

Si veda la sezione 86.1.

```

5720001 #include <unistd.h>
5720002 #include <stdlib.h>
5720003 #include <signal.h>
5720004 //-----
5720005 int
5720006 main (void)
5720007 {
5720008     pid_t pid;
5720009     //-----
5720010     pid = fork ();
5720011     if (pid == 0)
5720012     {
5720013         setuid ((uid_t) 10);
5720014         execve ("/bin/aaa", NULL, NULL);
5720015         exit (0);
5720016     }
5720017     //-----
5720018     pid = fork ();

```

```

5720019     if (pid == 0)
5720020     {
5720021         setuid ((uid_t) 11);
5720022         execve ("/bin/bbb", NULL, NULL);
5720023         exit (0);
5720024     }
5720025     // -----
5720026     while (1)
5720027     {
5720028         ; // Just loop, to consume CPU time: it must be
5720029         // killed manually.
5720030     }
5720031     return (0);
5720032 }

```

## 96.1.8 applic/chgrp.c

« Si veda la sezione 86.6.

```

5730001 #include <unistd.h>
5730002 #include <stdlib.h>
5730003 #include <sys/stat.h>
5730004 #include <sys/types.h>
5730005 #include <fcntl.h>
5730006 #include <errno.h>
5730007 #include <stdio.h>
5730008 #include <ctype.h>
5730009 #include <grp.h>
5730010 // -----
5730011 static void usage (void);
5730012 // -----
5730013 int
5730014 main (int argc, char *argv[], char *envp[])
5730015 {
5730016     char *group;
5730017     int gid;
5730018     struct group *grs;
5730019     struct stat file_status;
5730020     int a; // Argument index.
5730021     int status;
5730022     //
5730023     //
5730024     //
5730025     if (argc < 3)
5730026     {
5730027         usage ();
5730028         return (1);
5730029     }
5730030     //
5730031     // Get group id number.
5730032     //
5730033     group = argv[1];
5730034     if (isdigit (*group))
5730035     {
5730036         gid = atoi (group);
5730037     }
5730038     else
5730039     {
5730040         grs = getgrnam (group);
5730041         if (grs == NULL)
5730042         {
5730043             fprintf (stderr, "Unknown group \"%s!\n",
5730044                     group);
5730045             return (2);
5730046         }
5730047         gid = grs->gr_gid;
5730048     }
5730049     //
5730050     // Now we have the group id. Start scanning file
5730051     // names.
5730052     //
5730053     for (a = 2; a < argc; a++)
5730054     {
5730055         //
5730056         // Verify if the file exists, through the return
5730057         // value of
5730058         // 'stat()'. No other checks are made.
5730059         //
5730060         if (stat (argv[a], &file_status) == 0)
5730061         {
5730062             //
5730063             // Try to change ownership.
5730064             //
5730065             status = chown (argv[a], file_status.st_uid, gid);
5730066             if (status != 0)
5730067             {
5730068                 perror (NULL);

```

```

5730069         return (3);
5730070     }
5730071     }
5730072     else
5730073     {
5730074         fprintf (stderr,
5730075                 "File \"%s\" does not exist!\n",
5730076                 argv[a]);
5730077         continue;
5730078     }
5730079     }
5730080     //
5730081     // All done.
5730082     //
5730083     return (0);
5730084 }
5730085 // -----
5730086 // -----
5730087 static void
5730088 usage (void)
5730089 {
5730090     fprintf (stderr, "Usage:  chown GROUP|UID FILE...\n");
5730091     fprintf (stderr, "Example: chown group my_file\n");
5730092 }

```

## 96.1.9 applic/chmod.c

« Si veda la sezione 86.7.

```

5740001 #include <unistd.h>
5740002 #include <stdlib.h>
5740003 #include <sys/stat.h>
5740004 #include <sys/types.h>
5740005 #include <fcntl.h>
5740006 #include <errno.h>
5740007 #include <signal.h>
5740008 #include <stdio.h>
5740009 #include <sys/wait.h>
5740010 #include <stdio.h>
5740011 #include <string.h>
5740012 #include <limits.h>
5740013 #include <sys/os32.h>
5740014 // -----
5740015 static void usage (void);
5740016 // -----
5740017 int
5740018 main (int argc, char *argv[], char *envp[])
5740019 {
5740020     int status;
5740021     mode_t mode;
5740022     char *m; // Pointer inside the octal mode
5740023     // string.
5740024     int digit;
5740025     int a; // Argument index.
5740026     //
5740027     //
5740028     //
5740029     if (argc < 3)
5740030     {
5740031         usage ();
5740032         return (1);
5740033     }
5740034     //
5740035     // Get mode: must be the first argument.
5740036     //
5740037     for (m = argv[1]; *m != 0; m++)
5740038     {
5740039         digit = (*m - '0');
5740040         if (digit < 0 || digit > 7)
5740041         {
5740042             usage ();
5740043             return (2);
5740044         }
5740045         mode = mode * 8 + digit;
5740046     }
5740047     //
5740048     // System call for all the remaining arguments.
5740049     //
5740050     for (a = 2; a < argc; a++)
5740051     {
5740052         status = chmod (argv[a], mode);
5740053         if (status != 0)
5740054         {
5740055             perror (argv[a]);
5740056             return (3);
5740057         }
5740058     }

```

```

5740059 //
5740060 // All done.
5740061 //
5740062 return (0);
5740063 }
5740064
5740065 //-----
5740066 static void
5740067 usage (void)
5740068 {
5740069     fprintf (stderr, "Usage:  chmod OCTAL_MODE FILE...\n");
5740070     fprintf (stderr, "Example: chmod 0640 my_file\n");
5740071 }

```

### 96.1.10 applic/chown.c

« Si veda la sezione 86.8.

```

5750001 #include <unistd.h>
5750002 #include <stdlib.h>
5750003 #include <sys/stat.h>
5750004 #include <sys/types.h>
5750005 #include <fcntl.h>
5750006 #include <errno.h>
5750007 #include <stdio.h>
5750008 #include <ctype.h>
5750009 #include <pwd.h>
5750010 //-----
5750011 static void usage (void);
5750012 //-----
5750013 int
5750014 main (int argc, char *argv[], char *envp[])
5750015 {
5750016     char *user;
5750017     int uid;
5750018     struct passwd *pws;
5750019     struct stat file_status;
5750020     int a; // Argument index.
5750021     int status;
5750022     //
5750023     //
5750024     //
5750025     if (argc < 3)
5750026     {
5750027         usage ();
5750028         return (1);
5750029     }
5750030     //
5750031     // Get user id number.
5750032     //
5750033     user = argv[1];
5750034     if (isdigit (*user))
5750035     {
5750036         uid = atoi (user);
5750037     }
5750038     else
5750039     {
5750040         pws = getpwnam (user);
5750041         if (pws == NULL)
5750042         {
5750043             fprintf (stderr, "Unknown user \"%s\"\n", user);
5750044             return (2);
5750045         }
5750046         uid = pws->pw_uid;
5750047     }
5750048     //
5750049     // Now we have the user id. Start scanning file
5750050     // names.
5750051     //
5750052     for (a = 2; a < argc; a++)
5750053     {
5750054         //
5750055         // Verify if the file exists, through the return
5750056         // value of
5750057         // 'stat()'. No other checks are made.
5750058         //
5750059         if (stat (argv[a], &file_status) == 0)
5750060         {
5750061             //
5750062             // Try to change ownership.
5750063             //
5750064             status = chown (argv[a], uid, file_status.st_gid);
5750065             if (status != 0)
5750066             {
5750067                 perror (NULL);
5750068                 return (3);
5750069             }

```

```

5750070     }
5750071     else
5750072     {
5750073         fprintf (stderr,
5750074             "File \"%s\" does not exist!\n",
5750075             argv[a]);
5750076         continue;
5750077     }
5750078 }
5750079 //
5750080 // All done.
5750081 //
5750082 return (0);
5750083 }
5750084
5750085 //-----
5750086 static void
5750087 usage (void)
5750088 {
5750089     fprintf (stderr, "Usage:  chown USER|UID FILE...\n");
5750090     fprintf (stderr, "Example: chown user my_file\n");
5750091 }

```

### 96.1.11 applic/cp.c

« Si veda la sezione 86.9.

```

5760001 #include <sys/os32.h>
5760002 #include <sys/stat.h>
5760003 #include <sys/types.h>
5760004 #include <unistd.h>
5760005 #include <stdlib.h>
5760006 #include <fcntl.h>
5760007 #include <errno.h>
5760008 #include <signal.h>
5760009 #include <stdio.h>
5760010 #include <string.h>
5760011 #include <limits.h>
5760012 #include <libgen.h>
5760013 //-----
5760014 static void usage (void);
5760015 //-----
5760016 int
5760017 main (int argc, char *argv[], char *envp[])
5760018 {
5760019     char *source;
5760020     char *destination;
5760021     char *destination_full;
5760022     struct stat file_status;
5760023     int dest_is_a_dir = 0;
5760024     int a; // Argument index.
5760025     char path[PATH_MAX];
5760026     int fd_source = -1;
5760027     int fd_destination = -1;
5760028     char buffer_in[BUFSIZ];
5760029     char *buffer_out;
5760030     ssize_t count_in; // Read counter.
5760031     ssize_t count_out; // Write counter.
5760032     //
5760033     // There must be at least two arguments, plus the
5760034     // program name.
5760035     //
5760036     if (argc < 3)
5760037     {
5760038         usage ();
5760039         return (1);
5760040     }
5760041     //
5760042     // Select the last argument as the destination.
5760043     //
5760044     destination = argv[argc - 1];
5760045     //
5760046     // Check if it is a directory and save it in a flag.
5760047     //
5760048     if (stat (destination, &file_status) == 0)
5760049     {
5760050         if (S_ISDIR (file_status.st_mode))
5760051         {
5760052             dest_is_a_dir = 1;
5760053         }
5760054     }
5760055     //
5760056     // If there are more than two arguments, verify that
5760057     // the last
5760058     // one is a directory.
5760059     //
5760060     if (argc > 3)

```

```

5760061 {
5760062     if (!dest_is_a_dir)
5760063     {
5760064         usage ();
5760065         fprintf (stderr, "The destination \"%s\" ",
5760066                 destination);
5760067         fprintf (stderr, "is not a directory!\n");
5760068         return (1);
5760069     }
5760070 }
5760071 //
5760072 // Scan the arguments, excluded the last, that is
5760073 // the destination.
5760074 //
5760075 for (a = 1; a < (argc - 1); a++)
5760076 {
5760077     //
5760078     // Source.
5760079     //
5760080     source = argv[a];
5760081     //
5760082     // Verify access permissions.
5760083     //
5760084     if (access (source, R_OK) < 0)
5760085     {
5760086         perror (source);
5760087         continue;
5760088     }
5760089     //
5760090     // Destination.
5760091     //
5760092     // If it is a directory, the destination path
5760093     // must be corrected.
5760094     //
5760095     if (dest_is_a_dir)
5760096     {
5760097         path[0] = 0;
5760098         strcat (path, destination);
5760099         strcat (path, "/");
5760100         strcat (path, basename (source));
5760101         //
5760102         // Update the destination path.
5760103         //
5760104         destination_full = path;
5760105     }
5760106     else
5760107     {
5760108         destination_full = destination;
5760109     }
5760110     //
5760111     // Check if destination file exists.
5760112     //
5760113     if (stat (destination_full, &file_status) == 0)
5760114     {
5760115         fprintf (stderr,
5760116                 "The destination file, \"%s\", ",
5760117                 destination_full);
5760118         fprintf (stderr, "already exists!\n");
5760119         continue;
5760120     }
5760121     //
5760122     // Everything is ready for the copy.
5760123     //
5760124     fd_source = open (source, O_RDONLY);
5760125     if (fd_source < 0)
5760126     {
5760127         perror (source);
5760128         //
5760129         // Continue with the next file.
5760130         //
5760131         continue;
5760132     }
5760133     //
5760134     fd_destination = creat (destination_full, 0777);
5760135     if (fd_destination < 0)
5760136     {
5760137         perror (destination);
5760138         close (fd_source);
5760139         //
5760140         // Continue with the next file.
5760141         //
5760142         continue;
5760143     }
5760144     //
5760145     // Copy the data.
5760146     //
5760147     while (1)

```

```

5760148     {
5760149         count_in =
5760150         read (fd_source, buffer_in, (size_t) BUFSIZ);
5760151         if (count_in > 0)
5760152         {
5760153             for (buffer_out = buffer_in; count_in > 0; )
5760154             {
5760155                 count_out =
5760156                 write (fd_destination, buffer_out,
5760157                       (size_t) count_in);
5760158                 if (count_out < 0)
5760159                 {
5760160                     perror (destination);
5760161                     close (fd_source);
5760162                     close (fd_destination);
5760163                     return (3);
5760164                 }
5760165                 //
5760166                 // If not all data is written,
5760167                 // continue writing,
5760168                 // but change the buffer start
5760169                 // position and the
5760170                 // amount to be written.
5760171                 //
5760172                 buffer_out += count_out;
5760173                 count_in -= count_out;
5760174             }
5760175         }
5760176         else if (count_in < 0)
5760177         {
5760178             perror (source);
5760179             close (fd_source);
5760180             close (fd_destination);
5760181         }
5760182         else
5760183         {
5760184             break;
5760185         }
5760186     }
5760187     //
5760188     if (close (fd_source))
5760189     {
5760190         perror (source);
5760191     }
5760192     if (close (fd_destination))
5760193     {
5760194         perror (destination);
5760195         return (4);
5760196     }
5760197 }
5760198 //
5760199 // All done.
5760200 //
5760201 return (0);
5760202 }
5760203 //-----
5760204 static void
5760205 usage (void)
5760206 {
5760207     fprintf (stderr, "Usage: cp OLD_NAME NEW_NAME\n");
5760208     fprintf (stderr, "        cp FILE... DIRECTORY\n");
5760209 }
5760210 }

```

## 96.1.12 applic/crt0.mer.s

Si veda la sezione 84.5.2.

```

5770001 .extern main
5770002 .extern _stdio_stream_setup
5770003 .extern _dirent_directory_stream_setup
5770004 .extern _atexit_setup
5770005 .extern _environment_setup
5770006 .global startup
5770007 .global _data_end
5770008 #-----
5770009 # Please note that, all segment descriptors are already
5770010 # set from the scheduler, and there is also data inside
5770011 # the stack, so that the call to 'main()' function will
5770012 # result as expected.
5770013 #-----
5770014 # The following statement says that the code will start
5770015 # at "startup" label.
5770016 #-----
5770017 .section .text
5770018 #-----
5770019 startup:

```

```

5770020 #
5770021 # Jump after initial data.
5770022 #
5770023 jmp startup_code
5770024 #
5770025 filler:
5770026 #
5770027 # After four bytes, from the start, there is the
5770028 # magic number and other data.
5770029 #
5770030 .space (0x0004 - (filler - startup))
5770031 #
5770032 magic:
5770033 .quad 0x6F7333326170706C # os32appl
5770034 ;
5770035 doffset: #
5770036 .int _text_start # Data offset from start.
5770037 etext: #
5770038 .int _text_end # End of code
5770039 edata: #
5770040 .int _data_end # End of initialized data.
5770041 ebss: #
5770042 .int _bss_end # End of not initialized data.
5770043 stack_size: #
5770044 .int 0x8000 # Requested stack size. Every
5770045 # single application
5770046 # might change this value.
5770047 #
5770048 # At the next label, the work begins.
5770049 #
5770050 .align 4
5770051 startup_code:
5770052 #
5770053 # Before the call to the main function, it is
5770054 # necessary to extract the value to assign to the
5770055 # global variable 'environ'. It is described as
5770056 # 'char **environ' and should contain the same
5770057 # address pointed by 'envp'. To get this value,
5770058 # the stack is popped and then pushed again.
5770059 # Please recall that the stack was prepared from
5770060 # the process management, at the 'exec()' system
5770061 # call.
5770062 #
5770063 pop %eax # argc
5770064 pop %ebx # argv
5770065 pop %ecx # envp
5770066 mov %ecx, environ # Variable 'environ' comes from
5770067 # <unistd.h>.
5770068
5770069 push %ecx
5770070 push %ebx
5770071 push %eax
5770072 #
5770073 # Could it be enough? Of course not!
5770074 # To be able to handle the
5770075 # environment, it must be copied inside the table
5770076 # '_environment_table[[]]', that is defined inside
5770077 # <stdlib.h>.
5770078 # To copy the environment it is used the function
5770079 # '_environment_setup()', passing the 'envp'
5770080 # pointer.
5770081 #
5770082 push %ecx
5770083 call _environment_setup
5770084 add $4, %esp
5770085 #
5770086 # After the environment copy is done, the value for
5770087 # the traditional variable 'environ' is updated, to
5770088 # point to the new array of pointer.
5770089 # The updated value comes from variable
5770090 # '_environment', defined inside <stdlib.h>.
5770091 # Then, also the 'argv' contained inside
5770092 # the stack is replaced with the new value.
5770093 #
5770094 mov $_environment, %eax
5770095 mov %eax, environ
5770096 #
5770097 pop %eax # argc
5770098 pop %ebx # argv[[]]
5770099 pop %ecx # envp[[]]
5770100 mov $_environment, %ecx
5770101 push %ecx
5770102 push %ebx
5770103 push %eax
5770104 #
5770105 # Setup standard I/O streams and at-exit table.
5770106 #
5770107 call _stdio_stream_setup

```

```

5770107 call _dirent_directory_stream_setup
5770108 call _atexit_setup
5770109 #
5770110 # Call the main function. The arguments are
5770111 # already pushed inside the stack.
5770112 #
5770113 call main
5770114 #
5770115 # Save the return value at the symbol 'exit_value'.
5770116 #
5770117 mov %eax, exit_value
5770118 #
5770119 .align 4
5770120 halt:
5770121 #
5770122 pushl $2 # Size of message.
5770123 pushl %exit_value # Pointer to the message.
5770124 pushl $6 # SYS_EXIT
5770125 call sys
5770126 add $4, %esp
5770127 add $4, %esp
5770128 add $4, %esp
5770129 #
5770130 jmp halt
5770131 #
5770132 #-----
5770133 .align 4
5770134 .section .rodata
5770135 #
5770136 data_magic:
5770137 .quad 0x6F73333264617461 # os32data [1]
5770138 #
5770139 _data_end:
5770140 .int _bss_end
5770141 #
5770142 # [1] This is placed here just to be the same as the
5770143 # other file 'crt0.sep.s'.
5770144 # See the other file for an explanation.
5770145 #-----
5770146 .align 4
5770147 .section .data
5770148 #
5770149 exit_value:
5770150 .int 0x00000000
5770151 #-----
5770152 .align 4
5770153 .section .bss

```

### 96.1.13 applic/crt0.sep.s

Si veda la sezione [84.5.2](#).

```

5780001 .extern main
5780002 .extern _stdio_stream_setup
5780003 .extern _dirent_directory_stream_setup
5780004 .extern _atexit_setup
5780005 .extern _environment_setup
5780006 .global startup
5780007 .global _data_end
5780008 #-----
5780009 # Please note that, all segment descriptors are already
5780010 # set from the scheduler, and there is also data inside
5780011 # the stack, so that the call to 'main()' function will
5780012 # result as expected.
5780013 #-----
5780014 # The following statement says that the code will start
5780015 # at "startup" label.
5780016 #-----
5780017 .section .text
5780018 #-----
5780019 startup:
5780020 #
5780021 # Jump after initial data.
5780022 #
5780023 jmp startup_code
5780024 #
5780025 filler:
5780026 #
5780027 # After four bytes, from the start, there is the
5780028 # magic number and other data.
5780029 #
5780030 .space (0x0004 - (filler - startup))
5780031 #
5780032 magic:
5780033 .quad 0x6F7333326170706C # os32appl
5780034 ;
5780035 doffset: #

```

```

5780036 .int _text_end # Data offset from start: at
5780037 # the end of TEXT.
5780038 etext: #
5780039 .int _text_end # End of code
5780040 edata: #
5780041 .int _data_end # End of initialized data.
5780042 ebss: #
5780043 .int _bss_end # End of not initialized data.
5780044 stack_size: #
5780045 .int 0x8000 # Requested stack size. Every
5780046 # single application
5780047 # might change this value.
5780048 #
5780049 # At the next label, the work begins.
5780050 #
5780051 .align 4
5780052 startup_code:
5780053 #
5780054 # Before the call to the main function, it is
5780055 # necessary to extract the value to assign to the
5780056 # global variable 'environ'. It is described as
5780057 # 'char **environ' and should contain the same
5780058 # address pointed by 'envp'. To get this value, the
5780059 # stack is popped and then pushed again.
5780060 # Please recall that the stack was prepared from
5780061 # the process management, at the 'exec()' system
5780062 # call.
5780063 #
5780064 pop %eax # argc
5780065 pop %ebx # argv
5780066 pop %ecx # envp
5780067 mov %ecx, environ # Variable 'environ' comes from
5780068 # <unistd.h>.
5780069 push %ecx
5780070 push %ebx
5780071 push %eax
5780072 #
5780073 # Could it be enough? Of course not! To be able to
5780074 # handle the environment, it must be copied inside
5780075 # the table '_environment_table[]', that is
5780076 # defined inside <stdlib.h>.
5780077 # To copy the environment, it is used the function
5780078 # '_environment_setup()', passing the 'envp'
5780079 # pointer.
5780080 #
5780081 push %ecx
5780082 call _environment_setup
5780083 add $4, %esp
5780084 #
5780085 # After the environment copy is done, the value for
5780086 # the traditional variable 'environ' is updated,
5780087 # to point to the new array of pointer.
5780088 # The updated value comes from variable
5780089 # '_environment', defined inside <stdlib.h>.
5780090 # Then, also the 'argv' contained inside
5780091 # the stack is replaced with the new value.
5780092 #
5780093 mov $_environment, %eax
5780094 mov %eax, environ
5780095 #
5780096 pop %eax # argc
5780097 pop %ebx # argv[][]
5780098 pop %ecx # envp[][]
5780099 mov $_environment, %ecx
5780100 push %ecx
5780101 push %ebx
5780102 push %eax
5780103 #
5780104 # Setup standard I/O streams and at-exit table.
5780105 #
5780106 call _stdio_stream_setup
5780107 call _dirent_directory_stream_setup
5780108 call _atexit_setup
5780109 #
5780110 # Call the main function. The arguments are already
5780111 # pushed inside the stack.
5780112 #
5780113 call main
5780114 #
5780115 # Save the return value at the symbol 'exit_value'.
5780116 #
5780117 mov %eax, exit_value
5780118 #
5780119 .align 4
5780120 halt:
5780121 #
5780122 pushl $2 # Size of message.

```

```

5780123 pushl $exit_value # Pointer to the message.
5780124 pushl $6 # SYS_EXIT
5780125 call sys
5780126 add $4, %esp
5780127 add $4, %esp
5780128 add $4, %esp
5780129 #
5780130 jmp halt
5780131 #
5780132 #-----
5780133 .align 4
5780134 .section .rodata
5780135 #
5780136 data_magic:
5780137 .quad 0x6F73333264617461 # os32data [1]
5780138 #
5780139 _data_end:
5780140 .int _bss_end
5780141 #
5780142 # [1] This signature is just a place holder, at the
5780143 # beginning of the data segment, which starts at
5780144 # address 0x00000000. This is to avoid constant
5780145 # strings to be placed exactly at the beginning
5780146 # (and it happened so), where the address is
5780147 # equal to 'NULL'.
5780148 #-----
5780149 .align 4
5780150 .section .data
5780151 #
5780152 exit_value:
5780153 .int 0x00000000
5780154 #-----
5780155 .align 4
5780156 .section .bss

```

## 96.1.14 applic/date.c

&lt;&lt;

## Si veda la sezione 86.10.

```

5790001 #include <unistd.h>
5790002 #include <stdlib.h>
5790003 #include <errno.h>
5790004 #include <time.h>
5790005 #include <ctype.h>
5790006 //-----
5790007 static void usage (void);
5790008 //-----
5790009 int
5790010 main (int argc, char *argv[], char *envp[])
5790011 {
5790012     struct tm *timeptr;
5790013     char string[5];
5790014     time_t timer;
5790015     int length;
5790016     char *input;
5790017     int i;
5790018     int status;
5790019     //
5790020     // There can be at most an argument.
5790021     //
5790022     if (argc > 2)
5790023     {
5790024         usage ();
5790025         return (1);
5790026     }
5790027     //
5790028     // Check if there is no argument: must show the
5790029     // date.
5790030     //
5790031     if (argc == 1)
5790032     {
5790033         timer = time (NULL);
5790034         printf ("%s\n", ctime (&timer));
5790035         return (0);
5790036     }
5790037     //
5790038     // There is one argument and must be the date do
5790039     // set.
5790040     //
5790041     input = argv[1];
5790042     //
5790043     // First get current date, for default values.
5790044     //
5790045     timer = time (NULL);
5790046     timeptr = gmtime (&timer);
5790047     //
5790048     // Verify to have a correct input.

```

```

5790049 //
5790050 length = (int) strlen (input);
5790051 if (length == 8 || length == 10 || length == 12)
5790052 {
5790053     for (i = 0; i < length; i++)
5790054     {
5790055         if (!isdigit (input[i]))
5790056         {
5790057             usage ();
5790058             return (2);
5790059         }
5790060     }
5790061 }
5790062 else
5790063 {
5790064     printf ("input: \"%s\": length: %i\n", input, length);
5790065     usage ();
5790066     return (3);
5790067 }
5790068 //
5790069 // Select the month.
5790070 //
5790071 string[0] = input[0];
5790072 string[1] = input[1];
5790073 string[2] = '\0';
5790074 timeptr->tm_mon = atoi (string);
5790075 //
5790076 // Select the day.
5790077 //
5790078 string[0] = input[2];
5790079 string[1] = input[3];
5790080 string[2] = '\0';
5790081 timeptr->tm_mday = atoi (string);
5790082 //
5790083 // Select the hour.
5790084 //
5790085 string[0] = input[4];
5790086 string[1] = input[5];
5790087 string[2] = '\0';
5790088 timeptr->tm_hour = atoi (string);
5790089 //
5790090 // Select the minute.
5790091 //
5790092 string[0] = input[6];
5790093 string[1] = input[7];
5790094 string[2] = '\0';
5790095 timeptr->tm_min = atoi (string);
5790096 //
5790097 // Select the year: must verify if there is a
5790098 // century.
5790099 //
5790100 if (length == 12)
5790101 {
5790102     string[0] = input[8];
5790103     string[1] = input[9];
5790104     string[2] = input[10];
5790105     string[3] = input[11];
5790106     string[4] = '\0';
5790107     timeptr->tm_year = atoi (string);
5790108 }
5790109 else if (length == 10)
5790110 {
5790111     sprintf (string, "%04i", timeptr->tm_year);
5790112     string[2] = input[8];
5790113     string[3] = input[9];
5790114     string[4] = '\0';
5790115     timeptr->tm_year = atoi (string);
5790116 }
5790117 //
5790118 // Now convert to 'time_t'.
5790119 //
5790120 timer = mktime (timeptr);
5790121 //
5790122 // Save to the system.
5790123 //
5790124 status = stime (&timer);
5790125 if (status != 0)
5790126 {
5790127     perror (NULL);
5790128 }
5790129 //
5790130 return (0);
5790131 }
5790132 //-----
5790133 static void
5790134 usage (void)
5790135

```

```

5790136 {
5790137     fprintf (stderr, "Usage: date [MMDDHHMM[CC][YY]]\n");
5790138 }

```

## 96.1.15 applic/ed.c

Si veda la sezione 86.11.

```

5800001 //-----
5800002 // 2009.08.18
5800003 // Modified by Daniele Giacomini for 'os16', to
5800004 // harmonize with it, even, when possible, on coding
5800005 // style.
5800006 //
5800007 // The original was taken form ELKS sources:
5800008 // 'elkscmd/misc_utils/ed.c'.
5800009 //-----
5800010 //
5800011 // Copyright (c) 1993 by David I. Bell
5800012 // Permission is granted to use, distribute, or modify
5800013 // this source, provided that this copyright notice
5800014 // remains intact.
5800015 //
5800016 // The "ed" built-in command (much simplified)
5800017 //
5800018 //-----
5800019
5800020 #include <stdio.h>
5800021 #include <ctype.h>
5800022 #include <unistd.h>
5800023 #include <stdbool.h>
5800024 #include <string.h>
5800025 #include <stdlib.h>
5800026 #include <fcntl.h>
5800027 //-----
5800028 #define isoctal(ch) (((ch) >= '0') && ((ch) <= '7'))
5800029 #define USERSIZE 1024 /* max line length typed in
5800030 by user */
5800031 #define INITBUFSIZE 1024 /* initial buffer size */
5800032 //-----
5800033 typedef int num_t;
5800034 typedef int len_t;
5800035 //
5800036 // The following is the type definition of structure
5800037 // 'line_t', but the structure contains pointers to the
5800038 // same kind of type. With the compiler Bcc, it is the
5800039 // only way to declare it.
5800040 //
5800041 typedef struct line line_t;
5800042 //
5800043 struct line
5800044 {
5800045     line_t *next;
5800046     line_t *prev;
5800047     len_t len;
5800048     char data[1];
5800049 };
5800050 //
5800051 static line_t lines;
5800052 static line_t *curline;
5800053 static num_t curnum;
5800054 static num_t lastnum;
5800055 static num_t marks[26];
5800056 static bool dirty;
5800057 static char *filename;
5800058 static char searchstring[USERSIZE];
5800059 //
5800060 static char *bufbase;
5800061 static char *bufptr;
5800062 static len_t bufused;
5800063 static len_t bufsize;
5800064 //-----
5800065 static void docommands (void);
5800066 static void subcommand (char *cp, num_t num1, num_t num2);
5800067 static bool getnum (char **retcp, bool * rethavenum,
5800068 num_t * retnum);
5800069 static bool setcurnum (num_t num);
5800070 static bool initedit (void);
5800071 static void termedit (void);
5800072 static void adlines (num_t num);
5800073 static bool insertline (num_t num, char *data, len_t len);
5800074 static bool deletelines (num_t num1, num_t num2);
5800075 static bool printlines (num_t num1, num_t num2,
5800076 bool expandflag);
5800077 static bool writelines (char *file, num_t num1, num_t num2);
5800078 static bool readlines (char *file, num_t num);
5800079 static num_t searchlines (char *str, num_t num1,

```



```

580080         num_t num2);
580081 static len_t findstring (line_t * lp, char *str,
580082                        len_t len, len_t offset);
580083 static line_t *findline (num_t num);
580084 //-----
580085 // Main.
580086 //-----
580087 int
580088 main (int argc, char *argv[], char *envp[])
580089 {
580090     if (!litedit ())
580091         return (2);
580092     //
580093     if (argc > 1)
580094     {
580095         filename = strdup (argv[1]);
580096         if (filename == NULL)
580097         {
580098             fprintf (stderr, "No memory\n");
580099             termit ();
580100             return (1);
580101         }
580102         //
580103         if (!readlines (filename, 1))
580104         {
580105             termit ();
580106             return (0);
580107         }
580108         //
580109         if (lastnum)
580110             setcurnum (1);
580111         //
580112         dirty = false;
580113     }
580114     //
580115     docommands ();
580116     //
580117     termit ();
580118     return (0);
580119 }
580120
580121 //-----
580122 // Read commands until we are told to stop.
580123 //-----
580124 void
580125 docommands (void)
580126 {
580127     char *cp;
580128     int len;
580129     num_t num1;
580130     num_t num2;
580131     bool havel;
580132     bool have2;
580133     char buf[USERSIZE];
580134     //
580135     while (true)
580136     {
580137         printf (": ");
580138         fflush (stdout);
580139         //
580140         if (fgets (buf, sizeof (buf), stdin) == NULL)
580141         {
580142             return;
580143         }
580144         //
580145         len = strlen (buf);
580146         if (len == 0)
580147         {
580148             return;
580149         }
580150         //
580151         cp = &buf[len - 1];
580152         if (*cp != '\n')
580153         {
580154             fprintf (stderr, "Command line too long\n");
580155             do
580156             {
580157                 len = fgetc (stdin);
580158             }
580159             while ((len != EOF) && (len != '\n'));
580160             //
580161             continue;
580162         }
580163         //
580164         while ((cp > buf) && isblank (cp[-1]))
580165         {
580166             cp--;

```

```

580167     }
580168     //
580169     *cp = '\0';
580170     //
580171     cp = buf;
580172     //
580173     while (isblank (*cp))
580174     {
580175         // *cp++;
580176         cp++;
580177     }
580178     //
580179     havel = false;
580180     have2 = false;
580181     //
580182     if ((curnum == 0) && (lastnum > 0))
580183     {
580184         curnum = 1;
580185         curline = lines.next;
580186     }
580187     //
580188     if (!getnum (&cp, &havel, &num1))
580189     {
580190         continue;
580191     }
580192     //
580193     while (isblank (*cp))
580194     {
580195         cp++;
580196     }
580197     //
580198     if (*cp == ',')
580199     {
580200         cp++;
580201         if (!getnum (&cp, &have2, &num2))
580202         {
580203             continue;
580204         }
580205         //
580206         if (!havel)
580207         {
580208             num1 = 1;
580209         }
580210         if (!have2)
580211         {
580212             num2 = lastnum;
580213         }
580214         havel = true;
580215         have2 = true;
580216     }
580217     //
580218     if (!havel)
580219     {
580220         num1 = curnum;
580221     }
580222     if (!have2)
580223     {
580224         num2 = num1;
580225     }
580226     //
580227     // Command interpretation switch.
580228     //
580229     switch (*cp++)
580230     {
580231     case 'a':
580232         addlines (num1 + 1);
580233         break;
580234         //
580235     case 'c':
580236         deletelines (num1, num2);
580237         addlines (num1);
580238         break;
580239         //
580240     case 'd':
580241         deletelines (num1, num2);
580242         break;
580243         //
580244     case 'f':
580245         if (*cp && !isblank (*cp))
580246         {
580247             fprintf (stderr, "Bad file command\n");
580248             break;
580249         }
580250         //
580251         while (isblank (*cp))
580252         {
580253             cp++;

```

```

5800254     }
5800255     if (*cp == '\0')
5800256     {
5800257         if (filename)
5800258         {
5800259             printf ("\t%s\n", filename);
5800260         }
5800261         else
5800262         {
5800263             printf ("No filename\n");
5800264         }
5800265         break;
5800266     }
5800267     //
5800268     cp = strdup (cp);
5800269     //
5800270     if (cp == NULL)
5800271     {
5800272         fprintf (stderr, "No memory for filename\n");
5800273         break;
5800274     }
5800275     //
5800276     if (filename)
5800277     {
5800278         free (filename);
5800279     }
5800280     //
5800281     filename = cp;
5800282     break;
5800283     //
5800284     case 'i':
5800285         addlines (num1);
5800286         break;
5800287     //
5800288     case 'k':
5800289         while (isblank (*cp))
5800290         {
5800291             cp++;
5800292         }
5800293         //
5800294         if ((*cp < 'a' || *cp > 'a') || cp[1])
5800295         {
5800296             fprintf (stderr, "Bad mark name\n");
5800297             break;
5800298         }
5800299         //
5800300         marks[*cp - 'a'] = num2;
5800301         break;
5800302         //
5800303     case 'l':
5800304         printlines (num1, num2, true);
5800305         break;
5800306         //
5800307     case 'p':
5800308         printlines (num1, num2, false);
5800309         break;
5800310         //
5800311     case 'q':
5800312         while (isblank (*cp))
5800313         {
5800314             cp++;
5800315         }
5800316         //
5800317         if (havel || *cp)
5800318         {
5800319             fprintf (stderr, "Bad quit command\n");
5800320             break;
5800321         }
5800322         //
5800323         if (!dirty)
5800324         {
5800325             return;
5800326         }
5800327         //
5800328         printf ("Really quit? ");
5800329         fflush (stdout);
5800330         //
5800331         buf[0] = '\0';
5800332         fgets (buf, sizeof (buf), stdin);
5800333         cp = buf;
5800334         //
5800335         while (isblank (*cp))
5800336         {
5800337             cp++;
5800338         }
5800339         //
5800340         if ((*cp == 'y') || (*cp == 'Y'))

```

```

5800341     {
5800342         return;
5800343     }
5800344     //
5800345     break;
5800346     //
5800347     case 'r':
5800348         if (*cp && !isblank (*cp))
5800349         {
5800350             fprintf (stderr, "Bad read command\n");
5800351             break;
5800352         }
5800353         //
5800354         while (isblank (*cp))
5800355         {
5800356             cp++;
5800357         }
5800358         //
5800359         if (*cp == '\0')
5800360         {
5800361             fprintf (stderr, "No filename\n");
5800362             break;
5800363         }
5800364         //
5800365         if (!havel)
5800366         {
5800367             num1 = lastnum;
5800368         }
5800369         //
5800370         // Open the file and add to the buffer
5800371         // at the next line.
5800372         //
5800373         if (readlines (cp, num1 + 1))
5800374         {
5800375             //
5800376             // If the file open fails, just
5800377             // break the command.
5800378             //
5800379             break;
5800380         }
5800381         //
5800382         // Set the default file name, if no
5800383         // previous name is available.
5800384         //
5800385         if (filename == NULL)
5800386         {
5800387             filename = strdup (cp);
5800388         }
5800389         //
5800390         break;
5800391         //
5800392     case 's':
5800393         subcommand (cp, num1, num2);
5800394         break;
5800395         //
5800396     case 'w':
5800397         if (*cp && !isblank (*cp))
5800398         {
5800399             fprintf (stderr, "Bad write command\n");
5800400             break;
5800401         }
5800402         //
5800403         while (isblank (*cp))
5800404         {
5800405             cp++;
5800406         }
5800407         //
5800408         if (!havel)
5800409         {
5800410             num1 = 1;
5800411             num2 = lastnum;
5800412         }
5800413         //
5800414         // If the file name is not specified, use
5800415         // the
5800416         // default one.
5800417         //
5800418         if (*cp == '\0')
5800419         {
5800420             cp = filename;
5800421         }
5800422         //
5800423         // If even the default file name is not
5800424         // specified,
5800425         // tell it.
5800426         //
5800427         if (cp == NULL)

```

```

5800428     {
5800429         fprintf (stderr, "No file name specified\n");
5800430         break;
5800431     }
5800432     //
5800433     // Write the file.
5800434     //
5800435     writelines (cp, num1, num2);
5800436     //
5800437     break;
5800438     //
5800439     case 'z':
5800440         switch (*cp)
5800441         {
5800442             case '-':
5800443                 printlines (curnum - 21, curnum, false);
5800444                 break;
5800445             case '.':
5800446                 printlines (curnum - 11, curnum + 10, false);
5800447                 break;
5800448             default:
5800449                 printlines (curnum, curnum + 21, false);
5800450                 break;
5800451         }
5800452         break;
5800453     //
5800454     case '.':
5800455         if (havel)
5800456         {
5800457             fprintf (stderr, "No arguments allowed\n");
5800458             break;
5800459         }
5800460         printlines (curnum, curnum, false);
5800461         break;
5800462         //
5800463     case '-':
5800464         if (setcurnum (curnum - 1))
5800465         {
5800466             printlines (curnum, curnum, false);
5800467         }
5800468         break;
5800469         //
5800470     case '=':
5800471         printf ("%d\n", num1);
5800472         break;
5800473         //
5800474     case '\0':
5800475         if (havel)
5800476         {
5800477             printlines (num2, num2, false);
5800478             break;
5800479         }
5800480         //
5800481         if (setcurnum (curnum + 1))
5800482         {
5800483             printlines (curnum, curnum, false);
5800484         }
5800485         break;
5800486         //
5800487     default:
5800488         fprintf (stderr, "Unimplemented command\n");
5800489         break;
5800490     }
5800491 }
5800492 }
5800493
5800494 //-----
5800495 // Do the substitute command.
5800496 // The current line is set to the last substitution
5800497 // done.
5800498 //-----
5800499 void
5800500 subcommand (char *cp, num_t num1, num_t num2)
5800501 {
5800502     int delim;
5800503     char *oldstr;
5800504     char *newstr;
5800505     len_t oldlen;
5800506     len_t newlen;
5800507     len_t deltalen;
5800508     len_t offset;
5800509     line_t *lp;
5800510     line_t *nlp;
5800511     bool globalflag;
5800512     bool printflag;
5800513     bool didsub;
5800514     bool needprint;

```

```

5800515     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
5800516     {
5800517         fprintf (stderr, "Bad line range for substitute\n");
5800518         return;
5800519     }
5800520     //
5800521     //
5800522     globalflag = false;
5800523     printflag = false;
5800524     didsub = false;
5800525     needprint = false;
5800526     //
5800527     if (isblank (*cp) || (*cp == '\0'))
5800528     {
5800529         fprintf (stderr, "Bad delimiter for substitute\n");
5800530         return;
5800531     }
5800532     //
5800533     delim = *cp++;
5800534     oldstr = cp;
5800535     //
5800536     cp = strchr (cp, delim);
5800537     //
5800538     if (cp == NULL)
5800539     {
5800540         fprintf (stderr,
5800541             "Missing 2nd delimiter for " "substitute\n");
5800542         return;
5800543     }
5800544     //
5800545     *cp++ = '\0';
5800546     //
5800547     newstr = cp;
5800548     cp = strchr (cp, delim);
5800549     //
5800550     if (cp)
5800551     {
5800552         *cp++ = '\0';
5800553     }
5800554     else
5800555     {
5800556         cp = "";
5800557     }
5800558     while (*cp)
5800559     {
5800560         switch (*cp++)
5800561         {
5800562             case 'g':
5800563                 globalflag = true;
5800564                 break;
5800565             //
5800566             case 'p':
5800567                 printflag = true;
5800568                 break;
5800569             //
5800570             default:
5800571                 fprintf (stderr,
5800572                     "Unknown option for substitute\n");
5800573                 return;
5800574         }
5800575     }
5800576     //
5800577     if (*oldstr == '\0')
5800578     {
5800579         if (searchstring[0] == '\0')
5800580         {
5800581             fprintf (stderr, "No previous search string\n");
5800582             return;
5800583         }
5800584         oldstr = searchstring;
5800585     }
5800586     //
5800587     if (oldstr != searchstring)
5800588     {
5800589         strcpy (searchstring, oldstr);
5800590     }
5800591     //
5800592     lp = findline (num1);
5800593     if (lp == NULL)
5800594     {
5800595         return;
5800596     }
5800597     //
5800598     oldlen = strlen (oldstr);
5800599     newlen = strlen (newstr);
5800600     deltalen = newlen - oldlen;
5800601     offset = 0;

```

```

5800602 //
5800603 while (num1 <= num2)
5800604 {
5800605     offset = findstring (lp, oldstr, oldlen, offset);
5800606     if (offset < 0)
5800607     {
5800608         if (needprint)
5800609         {
5800610             printlines (num1, num1, false);
5800611             needprint = false;
5800612         }
5800613         //
5800614         offset = 0;
5800615         lp = lp->next;
5800616         num1++;
5800617         continue;
5800618     }
5800619     //
5800620     needprint = printflag;
5800621     didsub = true;
5800622     dirty = true;
5800623
5800624     // -----
5800625     // If the replacement string is the same size or
5800626     // shorter
5800627     // than the old string, then the substitution is
5800628     // easy.
5800629     // -----
5800630
5800631     if (deltalen <= 0)
5800632     {
5800633         memcpy (&lp->data[offset], newstr, newlen);
5800634         //
5800635         if (deltalen)
5800636         {
5800637             memcpy (&lp->data[offset + newlen],
5800638                 &lp->data[offset + oldlen],
5800639                 lp->len - offset - oldlen);
5800640             //
5800641             lp->len += deltalen;
5800642         }
5800643         //
5800644         offset += newlen;
5800645         //
5800646         if (globalflag)
5800647         {
5800648             continue;
5800649         }
5800650         //
5800651         if (needprint)
5800652         {
5800653             printlines (num1, num1, false);
5800654             needprint = false;
5800655         }
5800656         //
5800657         lp = lp->next;
5800658         num1++;
5800659         continue;
5800660     }
5800661
5800662     // -----
5800663     // The new string is larger, so allocate a new
5800664     // line structure and use that.
5800665     // Link it in place of the old line structure.
5800666     // -----
5800667
5800668     nlp =
5800669         (line_t *) malloc (sizeof (line_t) + lp->len +
5800670             deltalen);
5800671     //
5800672     if (nlp == NULL)
5800673     {
5800674         fprintf (stderr, "Cannot get memory for line\n");
5800675         return;
5800676     }
5800677     //
5800678     nlp->len = lp->len + deltalen;
5800679     //
5800680     memcpy (nlp->data, lp->data, offset);
5800681     //
5800682     memcpy (&nlp->data[offset], newstr, newlen);
5800683     //
5800684     memcpy (&nlp->data[offset + newlen],
5800685         &lp->data[offset + oldlen],
5800686         lp->len - offset - oldlen);
5800687     //
5800688     nlp->next = lp->next;

```

```

5800689     nlp->prev = lp->prev;
5800690     nlp->prev->next = nlp;
5800691     nlp->next->prev = nlp;
5800692     //
5800693     if (curline == lp)
5800694     {
5800695         curline = nlp;
5800696     }
5800697     //
5800698     free (lp);
5800699     lp = nlp;
5800700     //
5800701     offset += newlen;
5800702     //
5800703     if (globalflag)
5800704     {
5800705         continue;
5800706     }
5800707     //
5800708     if (needprint)
5800709     {
5800710         printlines (num1, num1, false);
5800711         needprint = false;
5800712     }
5800713     //
5800714     lp = lp->next;
5800715     num1++;
5800716     }
5800717     //
5800718     if (!didsub)
5800719     {
5800720         fprintf (stderr,
5800721             "No substitutions found for \"%s\"\n",
5800722             oldstr);
5800723     }
5800724     }
5800725
5800726     //-----
5800727     // Search a line for the specified string starting at
5800728     // the specified offset in the line. Returns the
5800729     // offset of the found string, or -1.
5800730     //-----
5800731     len_t
5800732     findstring (line_t * lp, char *str, len_t len, len_t offset)
5800733     {
5800734         len_t left;
5800735         char *cp;
5800736         char *ncp;
5800737         //
5800738         cp = &lp->data[offset];
5800739         left = lp->len - offset;
5800740         //
5800741         while (left >= len)
5800742         {
5800743             ncp = memchr (cp, *str, left);
5800744             if (ncp == NULL)
5800745             {
5800746                 return (len_t) - 1;
5800747             }
5800748             //
5800749             left -= (ncp - cp);
5800750             if (left < len)
5800751             {
5800752                 return (len_t) - 1;
5800753             }
5800754             //
5800755             cp = ncp;
5800756             if (memcmp (cp, str, len) == 0)
5800757             {
5800758                 return (len_t) (cp - lp->data);
5800759             }
5800760             //
5800761             cp++;
5800762             left--;
5800763         }
5800764         //
5800765         return (len_t) - 1;
5800766     }
5800767
5800768     //-----
5800769     // Add lines which are typed in by the user.
5800770     // The lines are inserted just before the specified
5800771     // line number.
5800772     // The lines are terminated by a line containing a
5800773     // single dot (ugly!), or by an end of file.
5800774     //-----
5800775     void

```

```

5800776 addlines (num_t num)
5800777 {
5800778     int len;
5800779     char buf[USERSIZE + 1];
5800780     //
5800781     while (fgets (buf, sizeof (buf), stdin))
5800782     {
5800783         if ((buf[0] == '.' && (buf[1] == '\n')
5800784             && (buf[2] == '\0'))
5800785         {
5800786             return;
5800787         }
5800788         //
5800789         len = strlen (buf);
5800790         //
5800791         if (len == 0)
5800792         {
5800793             return;
5800794         }
5800795         //
5800796         if (buf[len - 1] != '\n')
5800797         {
5800798             fprintf (stderr, "Line too long\n");
5800799             //
5800800             do
5800801             {
5800802                 len = fgetc (stdin);
5800803             }
5800804             while ((len != EOF) && (len != '\n'));
5800805             //
5800806             return;
5800807         }
5800808         //
5800809         if (!insertline (num++, buf, len))
5800810         {
5800811             return;
5800812         }
5800813     }
5800814 }
5800815
5800816 //-----
5800817 // Parse a line number argument if it is present. This
5800818 // is a sum or difference of numbers, '.', '$', 'x, or
5800819 // a search string.
5800820 // Returns true if successful (whether or not there was
5800821 // a number).
5800822 // Returns false if there was a parsing error, with a
5800823 // message output.
5800824 // Whether there was a number is returned indirectly,
5800825 // as is the number.
5800826 // The character pointer which stopped the scan is also
5800827 // returned.
5800828 //-----
5800829 static bool
5800830 getnum (char **retcp, bool * rethavenum, num_t * retnum)
5800831 {
5800832     char *cp;
5800833     char *str;
5800834     bool havenum;
5800835     num_t value;
5800836     num_t num;
5800837     num_t sign;
5800838     //
5800839     cp = *retcp;
5800840     havenum = false;
5800841     value = 0;
5800842     sign = 1;
5800843     //
5800844     while (true)
5800845     {
5800846         while (isblank (*cp))
5800847         {
5800848             cp++;
5800849         }
5800850         //
5800851         switch (*cp)
5800852         {
5800853             case '.':
5800854                 havenum = true;
5800855                 num = curnum;
5800856                 cp++;
5800857                 break;
5800858             //
5800859             case '$':
5800860                 havenum = true;
5800861                 num = lastnum;
5800862                 cp++;

```

```

5800863         break;
5800864         //
5800865         case '\\':
5800866             cp++;
5800867             if ((*cp < 'a') || (*cp > 'z'))
5800868             {
5800869                 fprintf (stderr, "Bad mark name\n");
5800870                 return false;
5800871             }
5800872             //
5800873             havenum = true;
5800874             num = marks[*cp++ - 'a'];
5800875             break;
5800876             //
5800877         case '/':
5800878             str = ++cp;
5800879             cp = strchr (str, '/');
5800880             if (cp)
5800881             {
5800882                 *cp++ = '\0';
5800883             }
5800884             else
5800885             {
5800886                 cp = "";
5800887             }
5800888             num = searchlines (str, curnum, lastnum);
5800889             if (num == 0)
5800890             {
5800891                 return false;
5800892             }
5800893             //
5800894             havenum = true;
5800895             break;
5800896             //
5800897         default:
5800898             if (!isdigit (*cp))
5800899             {
5800900                 *retcp = cp;
5800901                 *rethavenum = havenum;
5800902                 *retnum = value;
5800903                 return true;
5800904             }
5800905             //
5800906             num = 0;
5800907             while (isdigit (*cp))
5800908             {
5800909                 num = num * 10 + *cp++ - '0';
5800910             }
5800911             havenum = true;
5800912             break;
5800913         }
5800914         //
5800915         value += num * sign;
5800916         //
5800917         while (isblank (*cp))
5800918         {
5800919             cp++;
5800920         }
5800921         //
5800922         switch (*cp)
5800923         {
5800924             case '-':
5800925                 sign = -1;
5800926                 cp++;
5800927                 break;
5800928             //
5800929             case '+':
5800930                 sign = 1;
5800931                 cp++;
5800932                 break;
5800933             //
5800934             default:
5800935                 *retcp = cp;
5800936                 *rethavenum = havenum;
5800937                 *retnum = value;
5800938                 return true;
5800939         }
5800940     }
5800941 }
5800942
5800943 //-----
5800944 // Initialize everything for editing.
5800945 //-----
5800946 bool
5800947 initedit (void)
5800948 {
5800949     int i;

```

```

580050 //
580051 bufsize = INITBUFSIZE;
580052 bufbase = malloc (bufsize);
580053 //
580054 if (bufbase == NULL)
580055 {
580056     fprintf (stderr, "No memory for buffer\n");
580057     return false;
580058 }
580059 //
580060 bufptr = bufbase;
580061 bufused = 0;
580062 //
580063 lines.next = &lines;
580064 lines.prev = &lines;
580065 //
580066 curline = NULL;
580067 curnum = 0;
580068 lastnum = 0;
580069 dirty = false;
580070 filename = NULL;
580071 searchstring[0] = '\0';
580072 //
580073 for (i = 0; i < 26; i++)
580074 {
580075     marks[i] = 0;
580076 }
580077 //
580078 return true;
580079 }
580080
580081 //-----
580082 // Finish editing.
580083 //-----
580084 void
580085 termedit (void)
580086 {
580087     if (bufbase)
580088         free (bufbase);
580089     bufbase = NULL;
580090 //
580091     bufptr = NULL;
580092     bufsize = 0;
580093     bufused = 0;
580094 //
580095     if (filename)
580096         free (filename);
580097     filename = NULL;
580098 //
580099     searchstring[0] = '\0';
580100 //
580101     if (lastnum)
580102         deletelines (1, lastnum);
580103 //
580104     lastnum = 0;
580105     curnum = 0;
580106     curline = NULL;
580107 }
580108
580109 //-----
580110 // Read lines from a file at the specified line number.
580111 // Returns true if the file was successfully read.
580112 //-----
580113 bool
580114 readlines (char *file, num_t num)
580115 {
580116     int fd;
580117     int cc;
580118     len_t len;
580119     len_t linecount;
580120     len_t charcount;
580121     char *cp;
580122 //
580123     if ((num < 1) || (num > lastnum + 1))
580124     {
580125         fprintf (stderr, "Bad line for read\n");
580126         return false;
580127     }
580128 //
580129     fd = open (file, O_RDONLY);
580130     if (fd < 0)
580131     {
580132         perror (file);
580133         return false;
580134     }
580135 //
580136     bufptr = bufbase;

```

```

580137     bufused = 0;
580138     linecount = 0;
580139     charcount = 0;
580140 //
580141     printf ("\n%s\n", ", file);
580142     fflush (stdout);
580143 //
580144     do
580145     {
580146         cp = memchr (bufptr, '\n', bufused);
580147         if (cp)
580148         {
580149             len = (cp - bufptr) + 1;
580150             //
580151             if (!insertline (num, bufptr, len))
580152             {
580153                 close (fd);
580154                 return false;
580155             }
580156             //
580157             bufptr += len;
580158             bufused -= len;
580159             charcount += len;
580160             linecount++;
580161             num++;
580162             continue;
580163         }
580164         //
580165         if (bufptr != bufbase)
580166         {
580167             memcpy (bufbase, bufptr, bufused);
580168             bufptr = bufbase + bufused;
580169         }
580170         //
580171         if (bufused >= bufsize)
580172         {
580173             len = (bufsize + 3) / 2;
580174             cp = realloc (bufbase, len);
580175             if (cp == NULL)
580176             {
580177                 fprintf (stderr, "No memory for buffer\n");
580178                 close (fd);
580179                 return false;
580180             }
580181             //
580182             bufbase = cp;
580183             bufptr = bufbase + bufused;
580184             bufsize = len;
580185         }
580186         //
580187         cc = read (fd, bufptr, bufsize - bufused);
580188         bufused += cc;
580189         bufptr = bufbase;
580190     }
580191     while (cc > 0);
580192 //
580193     if (cc < 0)
580194     {
580195         perror (file);
580196         close (fd);
580197         return false;
580198     }
580199 //
580200     if (bufused)
580201     {
580202         if (!insertline (num, bufptr, bufused))
580203         {
580204             close (fd);
580205             return -1;
580206         }
580207         linecount++;
580208         charcount += bufused;
580209     }
580210 //
580211     close (fd);
580212 //
580213     printf ("%d lines%s, %d chars\n",
580214             linecount, (bufused ? " (incomplete)" : ""),
580215             charcount);
580216 //
580217     return true;
580218 }
580219
580220 //-----
580221 // Write the specified lines out to the specified file.
580222 // Returns true if successful, or false on an error
580223 // with a message output.

```

```

580124 //-----
580125 bool
580126 writelines (char *file, num_t num1, num_t num2)
580127 {
580128     int fd;
580129     line_t *lp;
580130     len_t linecount;
580131     len_t charcount;
580132     //
580133     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
580134     {
580135         fprintf (stderr, "Bad line range for write\n");
580136         return false;
580137     }
580138     //
580139     linecount = 0;
580140     charcount = 0;
580141     //
580142     fd = creat (file, 0666);
580143     if (fd < 0)
580144     {
580145         perror (file);
580146         return false;
580147     }
580148     //
580149     printf ("\n%s", ", file);
580150     fflush (stdout);
580151     //
580152     lp = findline (num1);
580153     if (lp == NULL)
580154     {
580155         close (fd);
580156         return false;
580157     }
580158     //
580159     while (num1++ <= num2)
580160     {
580161         if (write (fd, lp->data, lp->len) != lp->len)
580162         {
580163             perror (file);
580164             close (fd);
580165             return false;
580166         }
580167         //
580168         charcount += lp->len;
580169         linecount++;
580170         lp = lp->next;
580171     }
580172     //
580173     if (close (fd) < 0)
580174     {
580175         perror (file);
580176         return false;
580177     }
580178     //
580179     printf ("%d lines, %d chars\n", linecount, charcount);
580180     //
580181     return true;
580182 }
580183 //-----
580184 // Print lines in a specified range.
580185 // The last line printed becomes the current line.
580186 // If expandflag is true, then the line is printed
580187 // specially to show magic characters.
580188 //-----
580189 bool
580190 printlines (num_t num1, num_t num2, bool expandflag)
580191 {
580192     line_t *lp;
580193     unsigned char *cp;
580194     int ch;
580195     len_t count;
580196     //
580197     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
580198     {
580199         fprintf (stderr, "Bad line range for print\n");
580200         return false;
580201     }
580202     //
580203     lp = findline (num1);
580204     if (lp == NULL)
580205     {
580206         return false;
580207     }
580208     //
580209     //
580210     while (num1 <= num2)

```

```

580211 {
580212     if (!expandflag)
580213     {
580214         write (STDOUT_FILENO, lp->data, lp->len);
580215         setcurnum (num1++);
580216         lp = lp->next;
580217         continue;
580218     }
580219     //-----
580220     // Show control characters and characters with
580221     // the high bit set specially.
580222     //-----
580223     cp = (unsigned char *) lp->data;
580224     count = lp->len;
580225     //
580226     if ((count > 0) && (cp[count - 1] == '\n'))
580227     {
580228         count--;
580229     }
580230     //
580231     while (count-- > 0)
580232     {
580233         ch = *cp++;
580234         if (ch & 0x80)
580235         {
580236             fputs ("M-", stdout);
580237             ch &= 0x7F;
580238         }
580239         if (ch < ' ')
580240         {
580241             fputc ('^', stdout);
580242             ch += '@';
580243         }
580244         if (ch == 0x7E)
580245         {
580246             fputc ('^', stdout);
580247             ch = '?';
580248         }
580249         fputc (ch, stdout);
580250     }
580251     //
580252     fputs ("$\n", stdout);
580253     //
580254     setcurnum (num1++);
580255     lp = lp->next;
580256     //
580257     return true;
580258 }
580259 //-----
580260 // Insert a new line with the specified text.
580261 // The line is inserted so as to become the specified
580262 // line, thus pushing any existing and further lines
580263 // down one.
580264 // The inserted line is also set to become the current
580265 // line.
580266 // Returns true if successful.
580267 //-----
580268 bool
580269 insertline (num_t num, char *data, len_t len)
580270 {
580271     line_t *newlp;
580272     line_t *lp;
580273     //
580274     if ((num < 1) || (num > lastnum + 1))
580275     {
580276         fprintf (stderr, "Inserting at bad line number\n");
580277         return false;
580278     }
580279     //
580280     newlp = (line_t *) malloc (sizeof (line_t) + len - 1);
580281     if (newlp == NULL)
580282     {
580283         fprintf (stderr,
580284             "Failed to allocate memory for line\n");
580285         return false;
580286     }
580287     //
580288     memcpy (newlp->data, data, len);
580289     newlp->len = len;
580290     //
580291     if (num > lastnum)
580292     {
580293         lp = &lines;

```

```

5801298     }
5801299     else
5801300     {
5801301         lp = findline (num);
5801302         if (lp == NULL)
5801303         {
5801304             free ((char *) newlp);
5801305             return false;
5801306         }
5801307     }
5801308     //
5801309     newlp->next = lp;
5801310     newlp->prev = lp->prev;
5801311     lp->prev->next = newlp;
5801312     lp->prev = newlp;
5801313     //
5801314     lastnum++;
5801315     dirty = true;
5801316     //
5801317     return setcurnum (num);
5801318 }
5801319
5801320 //-----
5801321 // Delete lines from the given range.
5801322 //-----
5801323 bool
5801324 deletelines (num_t num1, num_t num2)
5801325 {
5801326     line_t *lp;
5801327     line_t *nlp;
5801328     line_t *plp;
5801329     num_t count;
5801330     //
5801331     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
5801332     {
5801333         fprintf (stderr, "Bad line numbers for delete\n");
5801334         return false;
5801335     }
5801336     //
5801337     lp = findline (num1);
5801338     if (lp == NULL)
5801339     {
5801340         return false;
5801341     }
5801342     //
5801343     if ((curnum >= num1) && (curnum <= num2))
5801344     {
5801345         if (num2 < lastnum)
5801346         {
5801347             setcurnum (num2 + 1);
5801348         }
5801349         else if (num1 > 1)
5801350         {
5801351             setcurnum (num1 - 1);
5801352         }
5801353         else
5801354         {
5801355             curnum = 0;
5801356         }
5801357     }
5801358     //
5801359     count = num2 - num1 + 1;
5801360     //
5801361     if (curnum > num2)
5801362     {
5801363         curnum -= count;
5801364     }
5801365     //
5801366     lastnum -= count;
5801367     //
5801368     while (count-- > 0)
5801369     {
5801370         nlp = lp->next;
5801371         plp = lp->prev;
5801372         plp->next = nlp;
5801373         nlp->prev = plp;
5801374         lp->next = NULL;
5801375         lp->prev = NULL;
5801376         lp->len = 0;
5801377         free (lp);
5801378         lp = nlp;
5801379     }
5801380     //
5801381     dirty = true;
5801382     //
5801383     return true;
5801384 }

```

```

5801385
5801386 //-----
5801387 // Search for a line which contains the specified
5801388 // string.
5801389 // If the string is NULL, then the previously searched
5801390 // for string is used. The currently searched for
5801391 // string is saved for future use.
5801392 // Returns the line number which matches, or 0 if there
5801393 // was no match with an error printed.
5801394 //-----
5801395 num_t
5801396 searchlines (char *str, num_t num1, num_t num2)
5801397 {
5801398     line_t *lp;
5801399     int len;
5801400     //
5801401     if ((num1 < 1) || (num2 > lastnum) || (num1 > num2))
5801402     {
5801403         fprintf (stderr, "Bad line numbers for search\n");
5801404         return 0;
5801405     }
5801406     //
5801407     if (*str == '\0')
5801408     {
5801409         if (searchstring[0] == '\0')
5801410         {
5801411             fprintf (stderr, "No previous search string\n");
5801412             return 0;
5801413         }
5801414         str = searchstring;
5801415     }
5801416     //
5801417     if (str != searchstring)
5801418     {
5801419         strcpy (searchstring, str);
5801420     }
5801421     //
5801422     len = strlen (str);
5801423     //
5801424     lp = findline (num1);
5801425     if (lp == NULL)
5801426     {
5801427         return 0;
5801428     }
5801429     //
5801430     while (num1 <= num2)
5801431     {
5801432         if (findstring (lp, str, len, 0) >= 0)
5801433         {
5801434             return num1;
5801435         }
5801436         //
5801437         num1++;
5801438         lp = lp->next;
5801439     }
5801440     //
5801441     fprintf (stderr, "Cannot find string \"%s\"\n", str);
5801442     //
5801443     return 0;
5801444 }
5801445
5801446 //-----
5801447 // Return a pointer to the specified line number.
5801448 //-----
5801449 line_t *
5801450 findline (num_t num)
5801451 {
5801452     line_t *lp;
5801453     num_t lnum;
5801454     //
5801455     if ((num < 1) || (num > lastnum))
5801456     {
5801457         fprintf (stderr,
5801458                 "Line number %d does not exist\n", num);
5801459         return NULL;
5801460     }
5801461     //
5801462     if (curnum <= 0)
5801463     {
5801464         curnum = 1;
5801465         curline = lines.next;
5801466     }
5801467     //
5801468     if (num == curnum)
5801469     {
5801470         return curline;
5801471     }

```



```

580472 //
580473 lp = curline;
580474 lnum = curnum;
580475 //
580476 if (num < (curnum / 2))
580477 {
580478     lp = lines.next;
580479     lnum = 1;
580480 }
580481 else if (num > ((curnum + lastnum) / 2))
580482 {
580483     lp = lines.prev;
580484     lnum = lastnum;
580485 }
580486 //
580487 while (lnum < num)
580488 {
580489     lp = lp->next;
580490     lnum++;
580491 }
580492 //
580493 while (lnum > num)
580494 {
580495     lp = lp->prev;
580496     lnum--;
580497 }
580498 //
580499 return lp;
580500 }
580501
580502 -----
580503 // Set the current line number.
580504 // Returns true if successful.
580505 -----
580506 bool
580507 setcurnum (num_t num)
580508 {
580509     line_t *lp;
580510     //
580511     lp = findline (num);
580512     if (lp == NULL)
580513     {
580514         return false;
580515     }
580516     //
580517     curnum = num;
580518     curline = lp;
580519     //
580520     return true;
580521 }
580522
580523 /* END CODE */

```

### 96.1.16 applic/getty.c

« Si veda la sezione 92.2.

```

581001 #include <unistd.h>
581002 #include <stdio.h>
581003 #include <stdlib.h>
581004 #include <signal.h>
581005 #include <sys/wait.h>
581006 #include <limits.h>
581007 #include <sys/os32.h>
581008 #include <fcntl.h>
581009 #include <stdio.h>
581010 -----
581011 int
581012 main (int argc, char *argv[], char *envp[])
581013 {
581014     char *device_name;
581015     int fdn;
581016     char *exec_argv[2];
581017     char **exec_envp;
581018     char buffer[BUFSIZ];
581019     ssize_t size_read;
581020     int status;
581021     //
581022     // The first argument is mandatory and must be a
581023     // console terminal.
581024     //
581025     device_name = argv[1];
581026     //
581027     // A console terminal is correctly selected (but it
581028     // is not checked
581029     // if it is a really available one).
581030     // Set as a process group leader.

```

```

581031 //
581032 setpgrp ();
581033 //
581034 // Open the terminal, that should become the
581035 // controlling terminal:
581036 // close the standard input and open the new
581037 // terminal (r/w).
581038 //
581039 close (0);
581040 fdn = open (device_name, O_RDWR);
581041 if (fdn < 0)
581042 {
581043     //
581044     // Cannot open terminal. A message should
581045     // appear, at least
581046     // to the current console.
581047     //
581048     perror (NULL);
581049     return (-1);
581050 }
581051 //
581052 // Reset terminal device permissions and ownership.
581053 //
581054 status = fchown (fdn, (uid_t) 0, (gid_t) 0);
581055 if (status != 0)
581056 {
581057     perror (NULL);
581058 }
581059 status = fchmod (fdn, 0644);
581060 if (status != 0)
581061 {
581062     perror (NULL);
581063 }
581064 //
581065 // The terminal is open and it should be already the
581066 // controlling
581067 // one: show '/etc/issue'. The same variable 'fdn'
581068 // is used, because
581069 // the controlling terminal will never be closed
581070 // (the exit syscall
581071 // will do it).
581072 //
581073 fdn = open ("/etc/issue", O_RDONLY);
581074 if (fdn > 0)
581075 {
581076     //
581077     // The file is present and is shown.
581078     //
581079     for (size_read = 1; size_read > 0;)
581080     {
581081         size_read =
581082             read (fdn, buffer, (size_t) (BUFSIZ - 1));
581083         if (size_read < 0)
581084         {
581085             break;
581086         }
581087         buffer[size_read] = '\0';
581088         printf ("%s", buffer);
581089     }
581090     close (fdn);
581091 }
581092 //
581093 // Show the terminal.
581094 //
581095 printf ("This is terminal %s\n", device_name);
581096 //
581097 // It is time to exec login: the environment is
581098 // inherited directly
581099 // from 'init'.
581100 //
581101 exec_argv[0] = "login";
581102 exec_argv[1] = NULL;
581103 exec_envp = envp;
581104 execve ("/bin/login", exec_argv, exec_envp);
581105 //
581106 // If 'execve()' returns, it is an error.
581107 //
581108 exit (-1);
581109 }

```

### 96.1.17 applic/http.c

« Si veda la sezione 92.3.

```

582001 #include <sys/stat.h>
582002 #include <sys/types.h>
582003 #include <unistd.h>

```

```

582004 #include <stdlib.h>
582005 #include <fcntl.h>
582006 #include <errno.h>
582007 #include <signal.h>
582008 #include <stdio.h>
582009 #include <string.h>
582010 #include <limits.h>
582011 #include <libgen.h>
582012 #include <arpa/inet.h>
582013 #include <sys/socket.h>
582014 #include <stdint.h>
582015 #include <stdbool.h>
582016 -----
582017 #define DEBUG 0
582018 static void usage (void);
582019 static int send_file (int sfdn2, const char *path);
582020 static int send_line (int sfdn2, const char *line);
582021 char buffer[BUFSIZ];
582022 char path_absolute[PATH_MAX];
582023 -----
582024 int
582025 main (int argc, char *argv[], char *envp[])
582026 {
582027     int opt;
582028     //extern char *optarg;           // not used.
582029     extern int optind;
582030     extern int optopt;
582031     //
582032     int status;
582033     int sfdn;
582034     int sfdn2;
582035     struct sockaddr_in sa_local;
582036     struct sockaddr_in sa_remote;
582037     socklen_t sa_remote_size = sizeof (struct sockaddr_in);
582038     ssize_t rcv_size;
582039     char *addr = "0.0.0.0";
582040     char *www = NULL;
582041     char *path = NULL;
582042     int port;
582043     bool request_read;
582044     int b;           // index inside the buffer string
582045     // buffer
582046     char *string = NULL;
582047     struct stat file_status;
582048     //
582049     // Check for options: no options at the moment.
582050     //
582051     while ((opt = getopt (argc, argv, ":")) != -1)
582052     {
582053         switch (opt)
582054         {
582055             case '?':
582056                 fprintf (stderr, "Unknown option -%c.\n", optopt);
582057                 usage ();
582058                 return (1);
582059                 break;
582060             case ':':
582061                 fprintf (stderr,
582062                     "Missing argument for option -%c\n",
582063                     optopt);
582064                 usage ();
582065                 return (1);
582066                 break;
582067             default:
582068                 fprintf (stderr,
582069                     "Getopt problem: "
582070                     "unknown option %c\n", opt);
582071                 usage ();
582072                 return (1);
582073         }
582074     }
582075     //
582076     // Arguments.
582077     //
582078     if (optind == (argc - 2))
582079     {
582080         //
582081         // There are exactly two arguments: the port and
582082         // the www root path.
582083         //
582084         port = atoi (argv[argc - 2]);
582085         www = argv[argc - 1];
582086     }
582087     else
582088     {
582089         //
582090         // Arguments wrong!

```

```

582091     //
582092     printf ("optind = %i = %s, argc = %i\n", optind,
582093         argv[optind], argc);
582094     usage ();
582095     return (2);
582096 }
582097 //
582098 // Set the local address.
582099 //
582100 sa_local.sin_family = AF_INET;
582101 sa_local.sin_port = htons (port);
582102 inet_pton (AF_INET, addr, &sa_local.sin_addr.s_addr);
582103 //
582104 // Open the socket.
582105 //
582106 sfdn = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
582107 if (sfdn < 0)
582108 {
582109     perror (NULL);
582110     return (3);
582111 }
582112 if (DEBUG)
582113 {
582114     printf ("HTTP: listening socket number "
582115         "is %i.\n", sfdn);
582116 }
582117 //
582118 // Set it listening: bind the local 'sa' location.
582119 //
582120 status = bind (sfdn, (struct sockaddr *) &sa_local,
582121     sizeof (sa_local));
582122 if (status < 0)
582123 {
582124     perror (NULL);
582125     close (sfdn);
582126     return (4);
582127 }
582128 //
582129 // Listen (TCP).
582130 //
582131 status = listen (sfdn, 1);
582132 if (status < 0)
582133 {
582134     perror (NULL);
582135     close (sfdn);
582136     return (5);
582137 }
582138 //
582139 // Accept connections, inside a loop.
582140 //
582141 while (1)
582142 {
582143     //
582144     // Accept.
582145     //
582146     if (DEBUG)
582147     {
582148         printf
582149             ("HTTP: listening socket number is %i.\n",
582150             sfdn);
582151     }
582152     //
582153     sfdn2 =
582154         accept (sfdn, (struct sockaddr *) &sa_remote,
582155             &sa_remote_size);
582156     //
582157     if (sfdn2 < 0)
582158     {
582159         perror (NULL);
582160         close (sfdn2);
582161         continue;
582162     }
582163     if (DEBUG)
582164     {
582165         printf
582166             ("HTTP: new connection with socket "
582167             "number %i.\n", sfdn2);
582168     }
582169     //
582170     // Define the socket non blocking.
582171     //
582172     status = fcntl (sfdn2, F_SETFL, O_NONBLOCK);
582173     if (status < 0)
582174     {
582175         perror (NULL);
582176         return (9);
582177     }

```

```

5820178 //
5820179 // Will read from the remote.
5820180 //
5820181 path_absolute[0] = 0;
5820182 request_read = 1;
5820183 while (request_read)
5820184 {
5820185 //
5820186 // Read a line from the remote side.
5820187 //
5820188 for (b = 0; b < (BUFSIZ - 2); b++, buffer[b] = 0)
5820189 {
5820190 //
5820191 // Read a single character from the
5820192 // remote side.
5820193 //
5820194 //
5820195 recv_size =
5820196 recv (sfdn2, &buffer[b], (size_t) 1, 0);
5820197 //
5820198 if (recv_size < 0)
5820199 {
5820200 if (errno == EAGAIN
5820201 || errno == EWOULDBLOCK)
5820202 {
5820203 b--;
5820204 continue;
5820205 }
5820206 else
5820207 {
5820208 perror (NULL);
5820209 close (sfdn2);
5820210 continue;
5820211 }
5820212 if (recv_size == 0)
5820213 {
5820214 //
5820215 // It is the end of stream, but
5820216 // should not happen.
5820217 //
5820218 buffer[b] = 0;
5820219 request_read = 0;
5820220 if (DEBUG)
5820221 {
5820222 printf ("HTTP: end of stream, "
5820223 "but should not "
5820224 "happen here! "
5820225 "%s\n", buffer);
5820226 }
5820227 break;
5820228 }
5820229 //
5820230 if (buffer[b] == '\r')
5820231 {
5820232 //
5820233 // Ignore CR.
5820234 //
5820235 b--;
5820236 continue;
5820237 }
5820238 //
5820239 if (buffer[b] == '\n')
5820240 {
5820241 //
5820242 // End of line.
5820243 //
5820244 buffer[b] = 0;
5820245 if (DEBUG)
5820246 {
5820247 printf ("HTTP: %s\n", buffer);
5820248 }
5820249 break;
5820250 }
5820251 }
5820252 //
5820253 // Was it the end of the header?
5820254 //
5820255 if (strlen (buffer) == 0)
5820256 {
5820257 //
5820258 // End of header.
5820259 //
5820260 request_read = 0;
5820261 break;
5820262 }
5820263 //
5820264 // We are reading the header: was it the GET

```

```

5820265 // command?
5820266 //
5820267 string = strtok (buffer, " ");
5820268 if (strncmp (string, "GET", 4) == 0)
5820269 {
5820270 //
5820271 // It is a GET: find the path.
5820272 //
5820273 path = strtok (NULL, " ");
5820274 strcat (path_absolute, www, PATH_MAX - 1);
5820275 strcat (path_absolute, path,
5820276 (PATH_MAX -
5820277 strlen (path_absolute) - 1));
5820278 }
5820279 }
5820280 //
5820281 // Verify to have received a 'GET' request.
5820282 //
5820283 if (strlen (path_absolute) == 0)
5820284 {
5820285 //
5820286 // There is no path inside the GET command;
5820287 // maybe there is
5820288 // no GET command either: 400
5820289 //
5820290 if (DEBUG)
5820291 {
5820292 printf ("HTTP: 400 Bad Request: "
5820293 "no path inside the GET "
5820294 "command.\n");
5820295 }
5820296 send_line (sfdn2, "HTTP/1.0 400 Bad Request\r\n");
5820297 send_line (sfdn2, "Content-Type: text/html\r\n");
5820298 send_line (sfdn2, "Content-Length: 26\r\n");
5820299 send_line (sfdn2, "\r\n");
5820300 send_line (sfdn2, "<H1>400 Bad Request</H1>\r\n");
5820301 }
5820302 //
5820303 // Verify the path.
5820304 //
5820305 if (stat (path_absolute, &file_status) != 0)
5820306 {
5820307 //
5820308 // The path inside the GET command does not
5820309 // exists: 404
5820310 //
5820311 if (DEBUG)
5820312 {
5820313 printf ("HTTP: 404 Not Found: "
5820314 "the path \"%s\" does not "
5820315 "exists.\n", path_absolute);
5820316 }
5820317 send_line (sfdn2, "HTTP/1.0 404 Not Found\r\n");
5820318 send_line (sfdn2, "Content-Type: text/html\r\n");
5820319 send_line (sfdn2, "Content-Length: 24\r\n");
5820320 send_line (sfdn2, "\r\n");
5820321 send_line (sfdn2, "<H1>404 Not Found</H1>\r\n");
5820322 }
5820323 else
5820324 {
5820325 //
5820326 // File exists: check the file type.
5820327 //
5820328 if (S_ISDIR (file_status.st_mode))
5820329 {
5820330 //
5820331 // Test to find 'index.html'.
5820332 //
5820333 strcat (path_absolute, "index.html",
5820334 (PATH_MAX -
5820335 strlen (path_absolute) - 1));
5820336 //
5820337 if (stat (path_absolute, &file_status) != 0)
5820338 {
5820339 //
5820340 // The index file inside the path
5820341 // requested
5820342 // does not exists: 404
5820343 //
5820344 if (DEBUG)
5820345 {
5820346 printf ("HTTP: 404 Not Found: "
5820347 "the path \"%s\" does "
5820348 "not exists.\n",
5820349 path_absolute);
5820350 }
5820351 send_line (sfdn2,

```

```

5820352         "HTTP/1.0 404 Not "
5820353         "Found\r\n");
5820354     send_line (sfdn2,
5820355         "Content-Type: "
5820356         "text/html\r\n");
5820357     send_line (sfdn2,
5820358         "Content-Length: 24\r\n");
5820359     send_line (sfdn2, "\r\n");
5820360     send_line (sfdn2,
5820361         "<H1>404 Not Found"
5820362         "<\/H1>\r\n");
5820363     }
5820364     }
5820365     //
5820366     // There is a file to send.
5820367     //
5820368     send_file (sfdn2, path_absolute);
5820369     }
5820370     //
5820371     // The socket 'sfdn2' might be already closed;
5820372     // if so, the variable was reset to zero.
5820373     //
5820374     if (sfdn2 != 0)
5820375         close (sfdn2);
5820376     buffer[0] = 0;
5820377     if (DEBUG)
5820378     {
5820379         printf
5820380         ("HTTP: connection closed: continue "
5820381         "listening.\n");
5820382     }
5820383     continue;
5820384     }
5820385     //
5820386     // All done.
5820387     //
5820388     close (sfdn);
5820389     return (0);
5820390 }
5820391 }
5820392 //-----
5820393 static int
5820394 send_file (int sfdn2, const char *path)
5820395 {
5820396     // size_t sent_size;
5820397     size_t file_size;
5820398     struct stat file_status;
5820399     char ascii_size[32];
5820400     int fdn;
5820401     char *buffer_in = buffer;
5820402     char *buffer_out;
5820403     ssize_t count_in;    // Read counter.
5820404     ssize_t count_out;  // Write counter.
5820405     //
5820406     if (sfdn2 == 0)
5820407         return (-1);
5820408     //
5820409     if (stat (path, &file_status) != 0)
5820410     {
5820411         perror (NULL);
5820412         close (sfdn2);
5820413         sfdn2 = 0;
5820414         return (-1);
5820415     }
5820416     //
5820417     file_size = file_status.st_size;
5820418     sprintf (ascii_size, "%i", file_size);
5820419     //
5820420     fdn = open (path, O_RDONLY);
5820421     if (fdn < 0)
5820422     {
5820423         if (DEBUG)
5820424         {
5820425             printf ("HTTP: 403 Forbidden: %s ", path);
5820426         }
5820427         perror (path);
5820428         send_line (sfdn2, "HTTP/1.0 403 Forbidden\r\n");
5820429         send_line (sfdn2, "Content-Type: text/html\r\n");
5820430         send_line (sfdn2, "Content-Length: 24\r\n");
5820431         send_line (sfdn2, "\r\n");
5820432         send_line (sfdn2, "<H1>404 Forbidden<\/H1>\r\n");
5820433         close (sfdn2);
5820434         sfdn2 = 0;
5820435         return (-1);
5820436     }
5820437     //
5820438     send_line (sfdn2, "HTTP/1.0 200 OK\r\n");

```

```

5820439     send_line (sfdn2, "Content-Type: text/html\r\n");
5820440     send_line (sfdn2, "Content-Length: ");
5820441     send_line (sfdn2, ascii_size);
5820442     send_line (sfdn2, "\r\n");
5820443     send_line (sfdn2, "\r\n");
5820444     //
5820445     // Copy the data.
5820446     //
5820447     while (1)
5820448     {
5820449         count_in = read (fdn, buffer_in, (size_t) BUFSIZ);
5820450         if (count_in > 0)
5820451         {
5820452             for (buffer_out = buffer_in; count_in > 0; )
5820453             {
5820454                 count_out = send (sfdn2, buffer_out,
5820455                     (size_t) count_in, 0);
5820456                 if (count_out < 0)
5820457                 {
5820458                     if (errno == EAGAIN
5820459                         || errno == EWOULDBLOCK)
5820460                     {
5820461                         continue;
5820462                     }
5820463                     else
5820464                     {
5820465                         fprintf (stderr,
5820466                             "[HTTP] cannot "
5820467                             "send 1!\n");
5820468                         perror (NULL);
5820469                         close (fdn);
5820470                         close (sfdn2);
5820471                         sfdn2 = 0;
5820472                         return (-1);
5820473                     }
5820474                 }
5820475                 //
5820476                 // If not all data is written, continue
5820477                 // writing, but change the buffer start
5820478                 // position and the
5820479                 // amount to be written.
5820480                 //
5820481                 buffer_out += count_out;
5820482                 count_in -= count_out;
5820483             }
5820484         }
5820485         else if (count_in < 0)
5820486         {
5820487             perror (path);
5820488             close (fdn);
5820489             close (sfdn2);
5820490             sfdn2 = 0;
5820491             return (-1);
5820492         }
5820493         else
5820494         {
5820495             break;
5820496         }
5820497     }
5820498     //
5820499     return (0);
5820500 }
5820501 }
5820502 //-----
5820503 static int
5820504 send_line (int sfdn2, const char *line)
5820505 {
5820506     size_t sent_size;
5820507     size_t line_size = strlen (line);
5820508     const char *start = line;
5820509     //
5820510     if (sfdn2 == 0)
5820511         return (-1);
5820512     //
5820513     while (1)
5820514     {
5820515         errno = 0;
5820516         sent_size =
5820517             send (sfdn2, start, (size_t) line_size, 0);
5820518         //
5820519         //
5820520         //
5820521         if (DEBUG)
5820522         {
5820523             printf
5820524             ("[HTTP] line_size=%i, sent_size=%i, "
5820525             "error=%i\n",

```

```

582026         (int) line_size, (int) sent_size, errno);
582027     }
582028     if (sent_size < 0)
582029     {
582030         if (errno == EAGAIN || errno == EWOULDBLOCK)
582031         {
582032             continue;
582033         }
582034         else
582035         {
582036             fprintf(stderr, "[HTTP] cannot send 2!\n");
582037             perror(NULL);
582038             close(sfdn2);
582039             sfdn2 = 0;
582040             return (-1);
582041         }
582042     }
582043     //
582044     //
582045     //
582046     if (sent_size < line_size)
582047     {
582048         start = &start[sent_size];
582049         line_size -= sent_size;
582050         //
582051         continue;
582052     }
582053     return (0);
582054 }
582055 }
582056
582057 //-----
582058 static void
582059 usage (void)
582060 {
582061     fprintf (stderr,
582062             "os32 http usage:\n"
582063             "\n"
582064             "http PORT WWW_ROOT_PATH\n"
582065             "\n"
582066             "PORT port number listening for "
582067             "connections"
582068             "\n"
582069             "WWW_ROOT_PATH root for the published "
582070             "documents." "\n");
582071 }

```

## 96.1.18 applic/init.c

&lt;

Si veda la sezione 92.4.

```

583001 #include <unistd.h>
583002 #include <stdio.h>
583003 #include <stdlib.h>
583004 #include <signal.h>
583005 #include <sys/wait.h>
583006 #include <limits.h>
583007 #include <sys/os32.h>
583008 #include <fcntl.h>
583009 #include <string.h>
583010 //-----
583011 #define RESPAWN_MAX      7
583012 #define COMMAND_MAX      100
583013 #define ARGUMENTS_MAX    32
583014 #define LINE_MAX         1024
583015 //-----
583016 int
583017 main (int argc, char *argv[], char *envp[])
583018 {
583019     //
583020     // 'init.c' has its own 'init.ort0.s' with a very
583021     // small stack
583022     // size. Remember to verify to have enough room for
583023     // the stack.
583024     //
583025     pid_t pid;
583026     int status;
583027     char *exec_argv[ARGUMENTS_MAX];
583028     int count;
583029     char *exec_envp[3];
583030     char buffer[LINE_MAX];
583031     int r;          // Respawn table index.
583032     int b;          // Buffer index.
583033     size_t size_read;
583034     char *inittab_id;
583035     char *inittab_runlevels;
583036     char *inittab_action;

```

```

583037 char *inittab_process;
583038 int eof;
583039 int fd;
583040 //
583041 // It follows a table for commands to be respawn.
583042 //
583043 struct
583044 {
583045     pid_t pid;
583046     char command[COMMAND_MAX];
583047     respawn[RESPAWN_MAX];
583048 }
583049 // -----
583050 signal (SIGHUP, SIG_IGN);
583051 signal (SIGINT, SIG_IGN);
583052 signal (SIGQUIT, SIG_IGN);
583053 signal (SIGILL, SIG_IGN);
583054 signal (SIGABRT, SIG_IGN);
583055 signal (SIGFPE, SIG_IGN);
583056 // signal (SIGKILL, SIG_IGN); Cannot ignore SIGKILL.
583057 signal (SIGSEGV, SIG_IGN);
583058 signal (SIGPIPE, SIG_IGN);
583059 signal (SIGALRM, SIG_IGN);
583060 signal (SIGTERM, SIG_IGN);
583061 // signal (SIGSTOP, SIG_IGN); Cannot ignore SIGSTOP.
583062 signal (SIGTSTP, SIG_IGN);
583063 signal (SIGCONT, SIG_IGN);
583064 signal (SIGTTIN, SIG_IGN);
583065 signal (SIGTTOU, SIG_IGN);
583066 signal (SIGUSR1, SIG_IGN);
583067 signal (SIGUSR2, SIG_IGN);
583068 // -----
583069 printf ("init\n");
583070 // heap_clear ();
583071 // process_info ();
583072 // -----
583073 //
583074 // Reset the 'respawn' table.
583075 //
583076 for (r = 0; r < RESPAWN_MAX; r++)
583077 {
583078     respawn[r].pid = 0;
583079     respawn[r].command[0] = 0;
583080     respawn[r].command[COMMAND_MAX - 1] = 0;
583081 }
583082 //
583083 // Read the '/etc/inittab' file.
583084 //
583085 fd = open ("/etc/inittab", O_RDONLY);
583086 //
583087 if (fd < 0)
583088 {
583089     perror ("Cannot open file '/etc/inittab'");
583090     exit (-1);
583091 }
583092 //
583093 //
583094 //
583095 for (eof = 0, r = 0; !eof && r < RESPAWN_MAX; r++)
583096 {
583097     for (b = 0; b < LINE_MAX; b++)
583098     {
583099         size_read = read (fd, &buffer[b], (size_t) 1);
583100         if (size_read <= 0)
583101         {
583102             buffer[b] = 0;
583103             eof = 1; // Close the read loop.
583104             break;
583105         }
583106         if (buffer[b] == '\n')
583107         {
583108             buffer[b] = 0;
583109             break;
583110         }
583111     }
583112     //
583113     // Remove comments: just replace '#' with '\0'.
583114     //
583115     for (b = 0; b < LINE_MAX; b++)
583116     {
583117         if (buffer[b] == '#')
583118         {
583119             buffer[b] = 0;
583120             break;
583121         }
583122     }
583123     //

```

```

5830124 // If the buffer is an empty string, just loop
5830125 // to next
5830126 // record.
5830127 //
5830128 if (strlen (buffer) == 0)
5830129 {
5830130     r--;
5830131     continue;
5830132 }
5830133 //
5830134 //
5830135 //
5830136 inittab_id = strtok (buffer, ":");
5830137 inittab_runlevels = strtok (NULL, ":");
5830138 inittab_action = strtok (NULL, ":");
5830139 inittab_process = strtok (NULL, ":");
5830140 //
5830141 // Only action 'respawn' is used.
5830142 //
5830143 if (strcmp (inittab_action, "respawn") == 0)
5830144 {
5830145     strncpy (respawn[r].command, inittab_process,
5830146             COMMAND_MAX);
5830147 }
5830148 else
5830149 {
5830150     r--;
5830151 }
5830152 }
5830153 //
5830154 //
5830155 //
5830156 close (fd);
5830157 //
5830158 // Define common environment.
5830159 //
5830160 exec_envp[0] = "PATH=/bin:/usr/bin:/sbin:/usr/sbin";
5830161 exec_envp[1] = "CONSOLE=/dev/console";
5830162 exec_envp[2] = NULL;
5830163 //
5830164 // Start processes.
5830165 //
5830166 for (r = 0; r < RESPAWN_MAX; r++)
5830167 {
5830168     if (strlen (respawn[r].command) > 0)
5830169     {
5830170         respawn[r].pid = fork ();
5830171         if (respawn[r].pid == 0)
5830172         {
5830173             exec_argv[0] =
5830174                 strtok (respawn[r].command, " \\t");
5830175             for (count = 1;
5830176                 count < (ARGUMENTS_MAX - 2); count++)
5830177             {
5830178                 exec_argv[count] = strtok (NULL, " \\t");
5830179                 if (exec_argv[count] == NULL)
5830180                 {
5830181                     break;
5830182                 }
5830183             }
5830184             //
5830185             // Last element must be NULL, even if
5830186             // there are more
5830187             // arguments than allowed.
5830188             //
5830189             exec_argv[count] = NULL;
5830190             //
5830191             // Run!
5830192             //
5830193             execve (exec_argv[0], exec_argv, exec_envp);
5830194             perror (NULL);
5830195             exit (0);
5830196         }
5830197     }
5830198 }
5830199 //
5830200 // Wait for the death of child.
5830201 //
5830202 while (1)
5830203 {
5830204     pid = wait (&status);
5830205     for (r = 0; r < RESPAWN_MAX; r++)
5830206     {
5830207         if (pid == respawn[r].pid)
5830208         {
5830209             //
5830210             // Run it again.

```

```

5830211 //
5830212 respawn[r].pid = fork ();
5830213 if (respawn[r].pid == 0)
5830214 {
5830215     exec_argv[0] =
5830216         strtok (respawn[r].command, " \\t");
5830217     for (count = 1;
5830218         count < (ARGUMENTS_MAX - 2); count++)
5830219     {
5830220         exec_argv[count] =
5830221             strtok (NULL, " \\t");
5830222         if (exec_argv[count] == NULL)
5830223         {
5830224             break;
5830225         }
5830226     }
5830227     //
5830228     // Last element must be NULL, even
5830229     // if there are more
5830230     // arguments than allowed.
5830231     //
5830232     exec_argv[count] = NULL;
5830233     //
5830234     // Run!
5830235     //
5830236     execve (exec_argv[0], exec_argv,
5830237            exec_envp);
5830238     exit (0);
5830239 }
5830240 break;
5830241 }
5830242 }
5830243 }
5830244 }

```

## 96.1.19 applic/ipconfig.c

Si veda la sezione 92.5.

```

5840001 #include <sys/os32.h>
5840002 #include <kernel/net.h>
5840003 #include <unistd.h>
5840004 #include <stdio.h>
5840005 #include <fcntl.h>
5840006 #include <unistd.h>
5840007 #include <stdlib.h>
5840008 -----
5840009 #define NET_BUFFER_MAX 1024 // [1]
5840010 //
5840011 // [1] Enough to be able to read important data from
5840012 // the 'net_table[]', without stack overflow.
5840013 // In fact, the table 'net_table[]' contains
5840014 // also the interface frames, and there is no sense
5840015 // to read a full item.
5840016 -----
5840017 int
5840018 main (int argc, char *argv[], char *envp[])
5840019 {
5840020     int fd;
5840021     ssize_t size_read;
5840022     char buffer[NET_BUFFER_MAX];
5840023     int n;
5840024     net_t *net_table_item;
5840025     char string[80];
5840026     //
5840027     //
5840028     // All options are ignored, at the moment.
5840029     //
5840030     //
5840031     //
5840032     // Open '/dev/kmem_net', to get the network
5840033     // interface table.
5840034     //
5840035     //
5840036     fd = open ("/dev/kmem_net", O_RDONLY);
5840037     if (fd < 0)
5840038     {
5840039         printf ("[%s] Cannot open \"/dev/kmem_net\" ",
5840040                argv[0]);
5840041         perror (NULL);
5840042         exit (0);
5840043     }
5840044     //
5840045     // Print header.
5840046     //
5840047     printf ("dev "
5840048            "address/mask "

```

```

584049      "mac          " "io      irq\n");
584050      //
584051      // Scan NET items and then print body.
584052      //
584053      for (n = 0; n < NET_MAX_DEVICES; n++)
584054          {
584055              lseek (fd, (off_t) n, SEEK_SET);
584056              size_read = read (fd, buffer, NET_BUFFER_MAX);
584057              if (size_read < NET_BUFFER_MAX)
584058                  {
584059                      printf
584060                      ("%s] Cannot read \"dev/kmem_net\" "
584061                       "item %i ", argv[0], n);
584062                      perror (NULL);
584063                      continue;
584064                  }
584065              net_table_item = (net_t *) buffer;
584066              if (net_table_item->type != NET_DEV_NULL)
584067                  {
584068                      sprintf (string, "net%i      ", n);
584069                      string[6] = '\0';
584070                      printf ("%s", string);
584071                      //
584072                      sprintf (string, "%i.%i.%i.%i/%i "
584073                               "          ",
584074                               net_table_item->ip >> 24 & 0x000000FF,
584075                               net_table_item->ip >> 16 & 0x000000FF,
584076                               net_table_item->ip >> 8 & 0x000000FF,
584077                               net_table_item->ip >> 0 & 0x000000FF,
584078                               net_table_item->m);
584079                      string[20] = '\0';
584080                      printf ("%s", string);
584081                      //
584082                      if (net_table_item->type & NET_DEV_ETH)
584083                          {
584084                              printf
584085                              ("%02x:%02x:%02x:%02x:%02x:%02x "
584086                               "0x%04x %i",
584087                               net_table_item->ethernet.mac[0],
584088                               net_table_item->ethernet.mac[1],
584089                               net_table_item->ethernet.mac[2],
584090                               net_table_item->ethernet.mac[3],
584091                               net_table_item->ethernet.mac[4],
584092                               net_table_item->ethernet.mac[5],
584093                               net_table_item->ethernet.base_io,
584094                               net_table_item->ethernet.irq);
584095                          }
584096                      printf ("\n");
584097                  }
584098          }
584099      close (fd);
584100      return (0);
584101      }

```

### 96.1.20 applic/kill.c

Si veda la sezione 86.12.

```

585001 #include <sys/os32.h>
585002 #include <sys/stat.h>
585003 #include <sys/types.h>
585004 #include <unistd.h>
585005 #include <stdlib.h>
585006 #include <fcntl.h>
585007 #include <errno.h>
585008 #include <signal.h>
585009 #include <stdio.h>
585010 #include <string.h>
585011 #include <limits.h>
585012 #include <libgen.h>
585013 //-----
585014 static void usage (void);
585015 //-----
585016 int
585017 main (int argc, char *argv[], char *envp[])
585018 {
585019     int signal;
585020     int pid;
585021     int a;          // Index inside arguments.
585022     int option_s = 0;
585023     int option_l = 0;
585024     int opt;
585025     extern char *optarg;
585026     extern int optopt;
585027     //
585028     // There must be at least an option, plus the
585029     // program name.

```

```

585030 //
585031 if (argc < 2)
585032     {
585033         usage ();
585034         return (1);
585035     }
585036 //
585037 // Check for options.
585038 //
585039 while ((opt = getopt (argc, argv, ":ls:")) != -1)
585040     {
585041         switch (opt)
585042             {
585043             case 'l':
585044                 option_l = 1;
585045                 break;
585046             case 's':
585047                 option_s = 1;
585048                 //
585049                 // In that case, there must be at least
585050                 // three arguments:
585051                 // the option, the signal and the process
585052                 // id.
585053                 //
585054                 if (argc < 4)
585055                     {
585056                         usage ();
585057                         return (1);
585058                     }
585059                 //
585060                 // Argument numbers are ok. Check the
585061                 // signal.
585062                 //
585063                 if (strcmp (optarg, "HUP") == 0)
585064                     {
585065                         signal = SIGHUP;
585066                     }
585067                 else if (strcmp (optarg, "INT") == 0)
585068                     {
585069                         signal = SIGINT;
585070                     }
585071                 else if (strcmp (optarg, "QUIT") == 0)
585072                     {
585073                         signal = SIGQUIT;
585074                     }
585075                 else if (strcmp (optarg, "ILL") == 0)
585076                     {
585077                         signal = SIGILL;
585078                     }
585079                 else if (strcmp (optarg, "ABRT") == 0)
585080                     {
585081                         signal = SIGABRT;
585082                     }
585083                 else if (strcmp (optarg, "FPE") == 0)
585084                     {
585085                         signal = SIGFPE;
585086                     }
585087                 else if (strcmp (optarg, "KILL") == 0)
585088                     {
585089                         signal = SIGKILL;
585090                     }
585091                 else if (strcmp (optarg, "SEGV") == 0)
585092                     {
585093                         signal = SIGSEGV;
585094                     }
585095                 else if (strcmp (optarg, "PIPE") == 0)
585096                     {
585097                         signal = SIGPIPE;
585098                     }
585099                 else if (strcmp (optarg, "ALRM") == 0)
585100                     {
585101                         signal = SIGALRM;
585102                     }
585103                 else if (strcmp (optarg, "TERM") == 0)
585104                     {
585105                         signal = SIGTERM;
585106                     }
585107                 else if (strcmp (optarg, "STOP") == 0)
585108                     {
585109                         signal = SIGSTOP;
585110                     }
585111                 else if (strcmp (optarg, "TSTP") == 0)
585112                     {
585113                         signal = SIGTSTP;
585114                     }
585115                 else if (strcmp (optarg, "CONT") == 0)
585116                     {

```

```

5850117         signal = SIGCONT;
5850118     }
5850119     else if (strcmp (optarg, "CHLD") == 0)
5850120     {
5850121         signal = SIGCHLD;
5850122     }
5850123     else if (strcmp (optarg, "TTIN") == 0)
5850124     {
5850125         signal = SIGTTIN;
5850126     }
5850127     else if (strcmp (optarg, "TTOU") == 0)
5850128     {
5850129         signal = SIGTTOU;
5850130     }
5850131     else if (strcmp (optarg, "USR1") == 0)
5850132     {
5850133         signal = SIGUSR1;
5850134     }
5850135     else if (strcmp (optarg, "USR2") == 0)
5850136     {
5850137         signal = SIGUSR2;
5850138     }
5850139     else
5850140     {
5850141         fprintf (stderr, "Unknown signal %s.\n",
5850142             optarg);
5850143         return (1);
5850144     }
5850145     break;
5850146 case '?':
5850147     fprintf (stderr, "Unknown option -%c.\n", optopt);
5850148     usage ();
5850149     return (1);
5850150     break;
5850151 case ':':
5850152     fprintf (stderr,
5850153         "Missing argument for option -%c\n",
5850154         optopt);
5850155     usage ();
5850156     return (1);
5850157     break;
5850158 default:
5850159     fprintf (stderr,
5850160         "Getopt problem: unknown option "
5850161         "%c\n", opt);
5850162     return (1);
5850163 }
5850164 }
5850165 //
5850166 //
5850167 //
5850168 if (option_l && option_s)
5850169 {
5850170     fprintf (stderr,
5850171         "Options \"-l\" and \"-s\" together ");
5850172     fprintf (stderr, "are incompatible.\n");
5850173     usage ();
5850174     return (1);
5850175 }
5850176 //
5850177 // Option "-l".
5850178 //
5850179 if (option_l)
5850180 {
5850181     printf ("HUP ");
5850182     printf ("INT ");
5850183     printf ("QUIT ");
5850184     printf ("ILL ");
5850185     printf ("ABRT ");
5850186     printf ("FPE ");
5850187     printf ("KILL ");
5850188     printf ("SEGV ");
5850189     printf ("PIPE ");
5850190     printf ("ALRM ");
5850191     printf ("TERM ");
5850192     printf ("STOP ");
5850193     printf ("TSTP ");
5850194     printf ("CONT ");
5850195     printf ("CHLD ");
5850196     printf ("TTIN ");
5850197     printf ("TTOU ");
5850198     printf ("USR1 ");
5850199     printf ("USR2 ");
5850200     printf ("\n");
5850201 }
5850202 //
5850203 // Option "-s".

```

```

5850204 //
5850205 if (option_s)
5850206 {
5850207     //
5850208     // Scan arguments.
5850209     //
5850210     for (a = 3; a < argc; a++)
5850211     {
5850212         //
5850213         // Get PID.
5850214         //
5850215         pid = atoi (argv[a]);
5850216         if (pid > 0)
5850217         {
5850218             //
5850219             // Kill.
5850220             //
5850221             if (kill (pid, signal) < 0)
5850222             {
5850223                 perror (argv[a]);
5850224             }
5850225         }
5850226     }
5850227     else
5850228     {
5850229         fprintf (stderr, "Invalid PID %s.", argv[a]);
5850230     }
5850231 }
5850232 //
5850233 // All done.
5850234 //
5850235 return (0);
5850236 }
5850237
5850238 -----
5850239 static void
5850240 usage (void)
5850241 {
5850242     fprintf (stderr, "Usage: kill -s SIGNAL_NAME PID...\n");
5850243     fprintf (stderr, "    kill -l\n");
5850244 }

```

## 96.1.21 applic/ln.c

Si veda la sezione 86.13.

«

```

5860001 #include <sys/os32.h>
5860002 #include <sys/stat.h>
5860003 #include <sys/types.h>
5860004 #include <unistd.h>
5860005 #include <stdlib.h>
5860006 #include <fcntl.h>
5860007 #include <errno.h>
5860008 #include <signal.h>
5860009 #include <stdio.h>
5860010 #include <string.h>
5860011 #include <limits.h>
5860012 #include <libgen.h>
5860013 -----
5860014 static void usage (void);
5860015 -----
5860016 int
5860017 main (int argc, char *argv[], char *envp[])
5860018 {
5860019     char *source;
5860020     char *destination;
5860021     char *new_destination;
5860022     struct stat file_status;
5860023     int dest_is_a_dir = 0;
5860024     int a; // Argument index.
5860025     char path[PATH_MAX];
5860026     //
5860027     // There must be at least two arguments, plus the
5860028     // program name.
5860029     //
5860030     if (argc < 3)
5860031     {
5860032         usage ();
5860033         return (1);
5860034     }
5860035     //
5860036     // Select the last argument as the destination.
5860037     //
5860038     destination = argv[argc - 1];
5860039     //
5860040     // Check if it is a directory and save it in a flag.
5860041     //

```



```

5860042 if (stat (destination, &file_status) == 0)
5860043 {
5860044     if (S_ISDIR (file_status.st_mode))
5860045     {
5860046         dest_is_a_dir = 1;
5860047     }
5860048 }
5860049 //
5860050 // If there are more than two arguments, verify that
5860051 // the last
5860052 // one is a directory.
5860053 //
5860054 if (argc > 3)
5860055 {
5860056     if (!dest_is_a_dir)
5860057     {
5860058         usage ();
5860059         fprintf (stderr, "The destination \"%s\" ",
5860060                 destination);
5860061         fprintf (stderr, "is not a directory!\n");
5860062         return (1);
5860063     }
5860064 }
5860065 //
5860066 // Scan the arguments, excluded the last, that is
5860067 // the destination.
5860068 //
5860069 for (a = 1; a < (argc - 1); a++)
5860070 {
5860071     //
5860072     // Source.
5860073     //
5860074     source = argv[a];
5860075     //
5860076     // Verify access permissions.
5860077     //
5860078     if (access (source, R_OK) < 0)
5860079     {
5860080         perror (source);
5860081         continue;
5860082     }
5860083     //
5860084     // Destination.
5860085     //
5860086     // If it is a directory, the destination path
5860087     // must be corrected.
5860088     //
5860089     if (dest_is_a_dir)
5860090     {
5860091         path[0] = 0;
5860092         strcat (path, destination);
5860093         strcat (path, "/");
5860094         strcat (path, basename (source));
5860095         //
5860096         // Update the destination path.
5860097         //
5860098         new_destination = path;
5860099     }
5860100     else
5860101     {
5860102         new_destination = destination;
5860103     }
5860104     //
5860105     // Check if destination file exists.
5860106     //
5860107     if (stat (new_destination, &file_status) == 0)
5860108     {
5860109         fprintf (stderr,
5860110                 "The destination file, \"%s\", ",
5860111                 new_destination);
5860112         fprintf (stderr, "already exists!\n");
5860113         continue;
5860114     }
5860115     //
5860116     // Everything is ready for the link.
5860117     //
5860118     if (link (source, new_destination) < 0)
5860119     {
5860120         perror (new_destination);
5860121         continue;
5860122     }
5860123 }
5860124 //
5860125 // All done.
5860126 //
5860127 return (0);
5860128 }

```

```

5860129 //-----
5860130 static void
5860131 usage (void)
5860132 {
5860133     fprintf (stderr, "Usage: ln OLD_NAME NEW_NAME\n");
5860134     fprintf (stderr, "       ln FILE... DIRECTORY\n");
5860135 }
5860136 //-----

```

## 96.1.22 applic/login.c

Si veda la sezione 86.14.

```

5870001 #include <unistd.h>
5870002 #include <stdlib.h>
5870003 #include <sys/stat.h>
5870004 #include <sys/types.h>
5870005 #include <fcntl.h>
5870006 #include <errno.h>
5870007 #include <unistd.h>
5870008 #include <signal.h>
5870009 #include <stdio.h>
5870010 #include <sys/wait.h>
5870011 #include <stdio.h>
5870012 #include <string.h>
5870013 #include <limits.h>
5870014 #include <stdint.h>
5870015 #include <sys/os32.h>
5870016 //-----
5870017 #define LOGIN_MAX      64
5870018 #define PASSWORD_MAX   64
5870019 #define HOME_MAX       64
5870020 #define LINE_MAX       1024
5870021 //-----
5870022 int
5870023 main (int argc, char *argv[], char *envp[])
5870024 {
5870025     char login[LOGIN_MAX];
5870026     char password[PASSWORD_MAX];
5870027     char buffer[LINE_MAX];
5870028     char *user_name;
5870029     char *user_password;
5870030     char *user_uid;
5870031     char *user_gid;
5870032     char *user_description;
5870033     char *user_home;
5870034     char *user_shell;
5870035     uid_t uid;
5870036     uid_t euid;
5870037     gid_t gid;
5870038     gid_t egid;
5870039     int fd;
5870040     ssize_t size_read;
5870041     int b; // Index inside buffer.
5870042     int loop;
5870043     char *exec_argv[2];
5870044     int status;
5870045     char *tty_path;
5870046     //
5870047     // Check if login is running correctly.
5870048     //
5870049     euid = geteuid ();
5870050     uid = getuid ();
5870051     //
5870052     // Check privileges.
5870053     //
5870054     if (!(uid == 0 && euid == 0))
5870055     {
5870056         printf
5870057             ("%s: can only run with root privileges!\n",
5870058              argv[0]);
5870059         exit (-1);
5870060     }
5870061     //
5870062     // Prepare arguments for the shell call.
5870063     //
5870064     exec_argv[0] = "--";
5870065     exec_argv[1] = NULL;
5870066     //
5870067     // Login.
5870068     //
5870069     while (1)
5870070     {
5870071         fd = open ("/etc/passwd", O_RDONLY);
5870072         //
5870073         if (fd < 0)
5870074         {

```

```

587075     perror ("Cannot open file '/etc/passwd'");
587076     exit (-1);
587077 }
587078 //
587079 printf ("Log in as \"root\" or \"user\" "
587080         "with password \"ciao\" :-)\n");
587081 input_line (login, "login: ", LOGIN_MAX,
587082            INPUT_LINE_ECHO);
587083 //
587084 //
587085 //
587086 loop = 1;
587087 while (loop)
587088 {
587089     for (b = 0; b < LINE_MAX; b++)
587090     {
587091         size_read = read (fd, &buffer[b], (size_t) 1);
587092         if (size_read <= 0)
587093         {
587094             buffer[b] = 0;
587095             loop = 0; // Close the middle
587096                 // loop.
587097             break;
587098         }
587099         if (buffer[b] == '\n')
587100         {
587101             buffer[b] = 0;
587102             break;
587103         }
587104     }
587105 //
587106 // Please notice that 'strtok()' does not
587107 // allow to have empty fields! If it finds
587108 // a ':', it will treat it as a single ':'.
587109 //
587110 user_name = strtok (buffer, ":");
587111 user_password = strtok (NULL, ":");
587112 user_uid = strtok (NULL, ":");
587113 user_gid = strtok (NULL, ":");
587114 user_description = strtok (NULL, ":");
587115 user_home = strtok (NULL, ":");
587116 user_shell = strtok (NULL, ":");
587117 //
587118 if (strcmp (user_name, login) == 0)
587119 {
587120     input_line (password, "password: ",
587121                PASSWORD_MAX, INPUT_LINE_HIDDEN);
587122 //
587123 // Compare passwords: empty passwords
587124 // are not allowed.
587125 //
587126 if (strcmp (user_password, password) == 0)
587127 {
587128     uid = atoi (user_uid);
587129     euid = uid;
587130     gid = atoi (user_gid);
587131     egid = gid;
587132 //
587133 // Find the controlling terminal and
587134 // change
587135 // property and access permissions.
587136 //
587137 tty_path = ttyname (STDIN_FILENO);
587138 if (tty_path != NULL)
587139 {
587140     status = chown (tty_path, uid, 0);
587141     if (status != 0)
587142     {
587143         perror (NULL);
587144     }
587145     status = chmod (tty_path, 0600);
587146     if (status != 0)
587147     {
587148         perror (NULL);
587149     }
587150 }
587151 //
587152 // Cd to the home directory, if
587153 // present.
587154 //
587155 status = chdir (user_home);
587156 if (status != 0)
587157 {
587158     perror (NULL);
587159 }
587160 //
587161 // Now change personality: first the

```

```

587062 // group,
587063 // otherwise, it would be not
587064 // possible to do
587065 // after changing the UID to an
587066 // unprivileged
587067 // one.
587068 //
587069 setgid (gid);
587070 setegid (egid);
587071 //
587072 setuid (uid);
587073 seteuid (euid);
587074 //
587075 // Run the shell, replacing the
587076 // login process; the
587077 // environment is taken from 'init'.
587078 //
587079 execve (user_shell, exec_argv, envp);
587080 exit (0);
587081 }
587082 //
587083 // Login failed: will try again.
587084 //
587085 loop = 0; // Close the middle loop.
587086 break;
587087 }
587088 }
587089 close (fd);
587090 }
587091 }

```

## 96.1.23 applic/ls.c

Si veda la sezione 86.15.

```

588001 #include <sys/stat.h>
588002 #include <sys/types.h>
588003 #include <unistd.h>
588004 #include <stdlib.h>
588005 #include <fcntl.h>
588006 #include <errno.h>
588007 #include <signal.h>
588008 #include <stdio.h>
588009 #include <string.h>
588010 #include <limits.h>
588011 #include <libgen.h>
588012 #include <dirent.h>
588013 #include <pwd.h>
588014 #include <grp.h>
588015 #include <time.h>
588016 //-----
588017 #define BUFFER_SIZE    131072
588018 #define LIST_SIZE      8000
588019 //-----
588020 static void usage (void);
588021 static int compare (const void *p1, const void *p2);
588022 //-----
588023 //
588024 // Static variables to avoid stack overflow.
588025 //
588026 static char buffer[BUFFER_SIZE];
588027 static char *list[LIST_SIZE];
588028 //-----
588029 int
588030 main (int argc, char *argv[], char *envp[])
588031 {
588032     int option_a = 0;
588033     int option_l = 0;
588034     int opt;
588035     // extern char *optarg; // not used.
588036     extern int optind;
588037     extern int optopt;
588038     struct stat file_status;
588039     DIR *dp;
588040     struct dirent *dir;
588041     int b; // Buffer index.
588042     int l; // List index.
588043     int len; // Name length.
588044     char *path = NULL;
588045     char pathname[PATH_MAX];
588046     struct passwd *pws;
588047     struct group *grs;
588048     struct tm *tms;
588049 //
588050 // Check for options.
588051 //
588052 while ((opt = getopt (argc, argv, "al")) != -1)

```

```

5880053 {
5880054     switch (opt)
5880055     {
5880056         case 'l':
5880057             option_l = 1;
5880058             break;
5880059         case 'a':
5880060             option_a = 1;
5880061             break;
5880062         case '?':
5880063             fprintf (stderr, "Unknown option -%c.\n", optopt);
5880064             usage ();
5880065             return (1);
5880066             break;
5880067         case ':':
5880068             fprintf (stderr,
5880069                 "Missing argument for option -%c\n",
5880070                 optopt);
5880071             usage ();
5880072             return (1);
5880073             break;
5880074         default:
5880075             fprintf (stderr,
5880076                 "Getopt problem: unknown option "
5880077                 "%c\n", opt);
5880078             return (1);
5880079     }
5880080 }
5880081 //
5880082 // If no arguments are present, at least the current
5880083 // directory is
5880084 // read.
5880085 //
5880086 if (optind == argc)
5880087 {
5880088     //
5880089     // There are no more arguments. Replace the
5880090     // program name,
5880091     // corresponding to 'argv[0]', with the current
5880092     // directory
5880093     // path string.
5880094     //
5880095     argv[0] = ".";
5880096     argc = 1;
5880097     optind = 0;
5880098 }
5880099 //
5880100 // This is a very simplified 'ls': if there is only
5880101 // a name
5880102 // and it is a directory, the directory content is
5880103 // taken as
5880104 // the new 'argv[]' array.
5880105 //
5880106 if (optind == (argc - 1))
5880107 {
5880108     //
5880109     // There is a request for a single name. Test if
5880110     // it exists
5880111     // and if it is a directory.
5880112     //
5880113     if (stat (argv[optind], &file_status) != 0)
5880114     {
5880115         fprintf (stderr,
5880116             "File \"%s\" does not exist!\n",
5880117             argv[optind]);
5880118         return (2);
5880119     }
5880120     //
5880121     if (S_ISDIR (file_status.st_mode))
5880122     {
5880123         //
5880124         // Save the directory inside the 'path'
5880125         // pointer.
5880126         //
5880127         path = argv[optind];
5880128         //
5880129         // Open the directory.
5880130         //
5880131         dp = opendir (argv[optind]);
5880132         if (dp == NULL)
5880133         {
5880134             perror (argv[optind]);
5880135             return (3);
5880136         }
5880137         //
5880138         // Read the directory and fill the buffer
5880139         // with names.

```

```

5880140     //
5880141     b = 0;
5880142     l = 0;
5880143     while ((dir = readdir (dp)) != NULL)
5880144     {
5880145         len = strlen (dir->d_name);
5880146         //
5880147         // Check if the buffer can hold it.
5880148         //
5880149         if ((b + len + 1) > BUFFER_SIZE)
5880150         {
5880151             fprintf (stderr, "not enough memory\n");
5880152             break;
5880153         }
5880154         //
5880155         // Consider the directory item only if
5880156         // there is
5880157         // a valid name. If it is empty, just
5880158         // ignore it.
5880159         //
5880160         if (len > 0)
5880161         {
5880162             strcpy (&buffer[b], dir->d_name);
5880163             list[l] = &buffer[b];
5880164             b += len + 1;
5880165             l++;
5880166         }
5880167     }
5880168     //
5880169     // Close the directory.
5880170     //
5880171     closedir (dp);
5880172     //
5880173     // Sort the list.
5880174     //
5880175     qsort (list, (size_t) l, sizeof (char *),
5880176         compare);
5880177     //
5880178     // Convert the directory list into a new
5880179     // 'argv[]' array,
5880180     // with a valid 'argc'. The variable
5880181     // 'optind' must be
5880182     // reset to the first element index, because
5880183     // there is
5880184     // no program name inside the new 'argv[]'
5880185     // at index zero.
5880186     //
5880187     argv = list;
5880188     argc = l;
5880189     optind = 0;
5880190 }
5880191 }
5880192 //
5880193 // Scan arguments, or list converted into 'argv[]'.
5880194 //
5880195 for (; optind < argc; optind++)
5880196 {
5880197     if (argv[optind][0] == '.')
5880198     {
5880199         //
5880200         // Current name starts with '.'.
5880201         //
5880202         if (!option_a)
5880203         {
5880204             //
5880205             // Do not show name starting with '.'.
5880206             //
5880207             continue;
5880208         }
5880209     }
5880210     //
5880211     // Build the pathname.
5880212     //
5880213     if (path == NULL)
5880214     {
5880215         strcpy (&pathname[0], argv[optind]);
5880216     }
5880217     else
5880218     {
5880219         strcpy (pathname, path);
5880220         strcat (pathname, "/");
5880221         strcat (pathname, argv[optind]);
5880222     }
5880223     //
5880224     // Check if file exists, reading status.
5880225     //
5880226     if (stat (pathname, &file_status) != 0)

```

```

5880227 {
5880228     fprintf (stderr,
5880229             "File \"%s\" does not exist!\n",
5880230             pathname);
5880231     return (2);
5880232 }
5880233 //
5880234 // Show file name.
5880235 //
5880236 if (option_l)
5880237 {
5880238     //
5880239     // Print the file type.
5880240     //
5880241     if (S_ISBLK (file_status.st_mode))
5880242         printf ("b");
5880243     else if (S_ISCHR (file_status.st_mode))
5880244         printf ("c");
5880245     else if (S_ISFIFO (file_status.st_mode))
5880246         printf ("p");
5880247     else if (S_ISREG (file_status.st_mode))
5880248         printf ("-");
5880249     else if (S_ISDIR (file_status.st_mode))
5880250         printf ("d");
5880251     else if (S_ISLNK (file_status.st_mode))
5880252         printf ("l");
5880253     else if (S_ISSOCK (file_status.st_mode))
5880254         printf ("s");
5880255     else
5880256         printf ("?");
5880257     //
5880258     // Print permissions.
5880259     //
5880260     if (S_IRUSR & file_status.st_mode)
5880261         printf ("r");
5880262     else
5880263         printf ("-");
5880264     if (S_IWUSR & file_status.st_mode)
5880265         printf ("w");
5880266     else
5880267         printf ("-");
5880268     if (S_IXUSR & file_status.st_mode)
5880269         printf ("x");
5880270     else
5880271         printf ("-");
5880272     if (S_IRGRP & file_status.st_mode)
5880273         printf ("r");
5880274     else
5880275         printf ("-");
5880276     if (S_IWGRP & file_status.st_mode)
5880277         printf ("w");
5880278     else
5880279         printf ("-");
5880280     if (S_IXGRP & file_status.st_mode)
5880281         printf ("x");
5880282     else
5880283         printf ("-");
5880284     if (S_IROTH & file_status.st_mode)
5880285         printf ("r");
5880286     else
5880287         printf ("-");
5880288     if (S_IWOTH & file_status.st_mode)
5880289         printf ("w");
5880290     else
5880291         printf ("-");
5880292     if (S_IXOTH & file_status.st_mode)
5880293         printf ("x");
5880294     else
5880295         printf ("-");
5880296     //
5880297     // Print links.
5880298     //
5880299     printf (" %3i", (int) file_status.st_nlink);
5880300     //
5880301     // Print owner.
5880302     //
5880303     pws = getpwuid (file_status.st_uid);
5880304     if (pws == NULL)
5880305     {
5880306         printf (" %i", (int) file_status.st_uid);
5880307     }
5880308     else
5880309     {
5880310         printf (" %s", pws->pw_name);
5880311     }
5880312     //
5880313     // Print group.

```

```

5880314     //
5880315     grs = getgrgid (file_status.st_gid);
5880316     if (grs == NULL)
5880317     {
5880318         printf (" %i", (int) file_status.st_gid);
5880319     }
5880320     else
5880321     {
5880322         printf (" %s", grs->gr_name);
5880323     }
5880324     //
5880325     // Print file size or device major-minor.
5880326     //
5880327     if (S_ISBLK (file_status.st_mode)
5880328         || S_ISCHR (file_status.st_mode))
5880329     {
5880330         printf (" %3i",
5880331                 (int) major (file_status.st_rdev));
5880332         printf (" %3i",
5880333                 (int) minor (file_status.st_rdev));
5880334     }
5880335     else
5880336     {
5880337         printf (" %8i", (int) file_status.st_size);
5880338     }
5880339     //
5880340     // Print modification date and time.
5880341     //
5880342     tms = localtime (&(file_status.st_mtime));
5880343     printf (" %4u-%02u-%02u %02u:%02u",
5880344             tms->tm_year, tms->tm_mon,
5880345             tms->tm_mday, tms->tm_hour, tms->tm_min);
5880346     //
5880347     // Print file name, but with no additional
5880348     // path.
5880349     //
5880350     printf (" %s\n", argv[optind]);
5880351 }
5880352 else
5880353 {
5880354     //
5880355     // Just show the file name and go to the
5880356     // next line.
5880357     //
5880358     printf ("%s\n", argv[optind]);
5880359 }
5880360 }
5880361 //
5880362 // All done.
5880363 //
5880364 return (0);
5880365 }
5880366
5880367 -----
5880368 static void
5880369 usage (void)
5880370 {
5880371     fprintf (stderr, "Usage: ls [OPTION] [FILE]...\n");
5880372 }
5880373
5880374 -----
5880375 static int
5880376 compare (const void *p1, const void *p2)
5880377 {
5880378     return (strcmp (*(char **) p1, *(char **) p2));
5880379 }

```

## 96.1.24 applic/man.c

Si veda la sezione 86.16.

```

5890001 #include <unistd.h>
5890002 #include <stdlib.h>
5890003 #include <errno.h>
5890004
5890005 #define MAX_LINES 20
5890006 #define MAX_COLUMNS 80
5890007
5890008 static char *man_page_directory = "/usr/share/man";
5890009
5890010 static void usage (void);
5890011 static FILE *open_man_page (int section, char *name);
5890012 static void build_path_name (int section, char *name,
5890013                             char *path);
5890014
5890015 int
5890016 main (int argc, char *argv[], char *envp[])

```



```

5890191         fclose (fp);
5890192         return (0);
5890193     }
5890194     //
5890195     // Backspace to overwrite '--More--' and the
5890196     // character
5890197     // pressed.
5890198     //
5890199     printf
5890200     (" \b \b\b \b\b \b\b \b\b "
5890201      "\b\b \b\b \b\b \b\b \b\b");
5890202     //
5890203     // Reset line/column counters.
5890204     //
5890205     column = 1;
5890206     line = 1;
5890207 }
5890208 //
5890209 // Close the file pointer if it is still open.
5890210 //
5890211 if (fp != NULL)
5890212 {
5890213     fclose (fp);
5890214 }
5890215 //
5890216 return (0);
5890217 }
5890218
5890219 -----
5890220 static void
5890221 usage (void)
5890222 {
5890223     fprintf (stderr, "Usage: man [SECTION] NAME\n");
5890224 }
5890225
5890226 -----
5890227 FILE *
5890228 open_man_page (int section, char *name)
5890229 {
5890230     FILE *fp;
5890231     char path[PATH_MAX];
5890232     struct stat file_status;
5890233     //
5890234     //
5890235     //
5890236     if (section > 0)
5890237     {
5890238         build_path_name (section, name, path);
5890239         //
5890240         // Check if file exists.
5890241         //
5890242         if (stat (path, &file_status) != 0)
5890243         {
5890244             fprintf (stderr,
5890245                      "Man page %s(%i) does not exist!\n",
5890246                      name, section);
5890247             return (NULL);
5890248         }
5890249     }
5890250     else
5890251     {
5890252         //
5890253         // Must try a section.
5890254         //
5890255         for (section = 1; section < 9; section++)
5890256         {
5890257             build_path_name (section, name, path);
5890258             //
5890259             // Check if file exists.
5890260             //
5890261             if (stat (path, &file_status) == 0)
5890262             {
5890263                 //
5890264                 // Found.
5890265                 //
5890266                 break;
5890267             }
5890268         }
5890269     }
5890270     //
5890271     // Check if a file was found.
5890272     //
5890273     if (section < 9)
5890274     {
5890275         fp = fopen (path, "r");
5890276         //
5890277         if (fp == NULL)

```

```

5890278     {
5890279         //
5890280         // Error opening file.
5890281         //
5890282         perror (path);
5890283         return (NULL);
5890284     }
5890285     else
5890286     {
5890287         //
5890288         // Opened right.
5890289         //
5890290         return (fp);
5890291     }
5890292 }
5890293 else
5890294 {
5890295     fprintf (stderr, "Man page %s does not exist!\n",
5890296             name);
5890297     return (NULL);
5890298 }
5890299 }
5890300
5890301 -----
5890302 void
5890303 build_path_name (int section, char *name, char *path)
5890304 {
5890305     char string_section[10];
5890306     //
5890307     // Convert the section number into a string.
5890308     //
5890309     sprintf (string_section, "%i", section);
5890310     //
5890311     // Prepare the path to the man file.
5890312     //
5890313     path[0] = 0;
5890314     strcat (path, man_page_directory);
5890315     strcat (path, "/");
5890316     strcat (path, name);
5890317     strcat (path, ".");
5890318     strcat (path, string_section);
5890319 }

```

### 96.1.25 applic/mkdir.c

Si veda la sezione 86.17.

```

5900001 #include <sys/os32.h>
5900002 #include <sys/stat.h>
5900003 #include <sys/types.h>
5900004 #include <unistd.h>
5900005 #include <stdlib.h>
5900006 #include <fcntl.h>
5900007 #include <errno.h>
5900008 #include <signal.h>
5900009 #include <stdio.h>
5900010 #include <string.h>
5900011 #include <limits.h>
5900012 #include <libgen.h>
5900013 -----
5900014 static int mkdir_parents (const char *path, mode_t mode);
5900015 static void usage (void);
5900016 -----
5900017 int
5900018 main (int argc, char *argv[], char *envp[])
5900019 {
5900020     sysmsg_uarea_t msg;
5900021     int status;
5900022     mode_t mode = 0;
5900023     int m; // Index inside mode argument.
5900024     int digit;
5900025     char **dir;
5900026     int d; // Directory index.
5900027     int option_p = 0;
5900028     int option_m = 0;
5900029     int opt;
5900030     extern char *optarg;
5900031     extern int optind;
5900032     extern int optopt;
5900033     //
5900034     // There must be at least an argument, plus the
5900035     // program name.
5900036     //
5900037     if (argc < 2)
5900038     {
5900039         usage ();
5900040         return (1);

```

```

5900041     }
5900042     //
5900043     // Check for options, starting from 'p'. The 'dir'
5900044     // pointer is used
5900045     // to calculate the argument pointer to the first
5900046     // directory [1].
5900047     // The macroinstruction 'max()' is declared inside
5900048     // <sys/os32.h>
5900049     // and does the expected thing.
5900050     //
5900051     while ((opt = getopt (argc, argv, ":pm:")) != -1)
5900052     {
5900053         switch (opt)
5900054         {
5900055             case 'm':
5900056                 option_m = 1;
5900057                 for (m = 0; m < strlen (optarg); m++)
5900058                 {
5900059                     digit = (optarg[m] - '0');
5900060                     if (digit < 0 || digit > 7)
5900061                     {
5900062                         usage ();
5900063                         return (2);
5900064                     }
5900065                     mode = mode * 8 + digit;
5900066                 }
5900067                 break;
5900068             case 'p':
5900069                 option_p = 1;
5900070                 break;
5900071             case '?':
5900072                 printf ("Unknown option -%c.\n", optopt);
5900073                 usage ();
5900074                 return (1);
5900075                 break;
5900076             case ':':
5900077                 printf ("Missing argument for option -%c\n",
5900078                     optopt);
5900079                 usage ();
5900080                 return (2);
5900081                 break;
5900082             default:
5900083                 printf
5900084                     ("Getopt problem: unknown option %c\n", opt);
5900085                 return (3);
5900086         }
5900087     }
5900088     //
5900089     dir = argv + optind;
5900090     //
5900091     // Check if the mode is to be set to a default
5900092     // value.
5900093     //
5900094     if (!option_m)
5900095     {
5900096         //
5900097         // Default mode.
5900098         //
5900099         sys (SYS_UAREA, &msg, (sizeof msg));
5900100         mode = 0777 & ~msg.umask;
5900101     }
5900102     //
5900103     // Directory creation.
5900104     //
5900105     for (d = 0; dir[d] != NULL; d++)
5900106     {
5900107         if (option_p)
5900108         {
5900109             status = mkdir_parents (dir[d], mode);
5900110             if (status != 0)
5900111             {
5900112                 perror (dir[d]);
5900113                 return (3);
5900114             }
5900115         }
5900116         else
5900117         {
5900118             status = mkdir (dir[d], mode);
5900119             if (status != 0)
5900120             {
5900121                 perror (dir[d]);
5900122                 return (4);
5900123             }
5900124         }
5900125     }
5900126     //
5900127     // All done.

```

```

5900128     //
5900129     return (0);
5900130 }
5900131
5900132 -----
5900133 static int
5900134 mkdir_parents (const char *path, mode_t mode)
5900135 {
5900136     char path_copy[PATH_MAX];
5900137     char *path_parent;
5900138     struct stat fst;
5900139     int status;
5900140     //
5900141     // Check if the path is empty.
5900142     //
5900143     if (path == NULL || strlen (path) == 0)
5900144     {
5900145         //
5900146         // Recursion ends here.
5900147         //
5900148         return (0);
5900149     }
5900150     //
5900151     // Check if it does already exists.
5900152     //
5900153     status = stat (path, &fst);
5900154     if (status == 0 && fst.st_mode & S_IFDIR)
5900155     {
5900156         //
5900157         // The path exists and is a directory.
5900158         //
5900159         return (0);
5900160     }
5900161     else if (status == 0 && !(fst.st_mode & S_IFDIR))
5900162     {
5900163         //
5900164         // The path exists but is not a directory.
5900165         //
5900166         errno = ENOTDIR; // Not a directory.
5900167         return (-1);
5900168     }
5900169     //
5900170     // Get the directory path.
5900171     //
5900172     strncpy (path_copy, path, PATH_MAX);
5900173     path_parent = dirname (path_copy);
5900174     //
5900175     // If it is '.', or '/', the recursion is
5900176     // terminated.
5900177     //
5900178     if (strcmp (path_parent, ".", PATH_MAX) == 0 ||
5900179         strcmp (path_parent, "/", PATH_MAX) == 0)
5900180     {
5900181         return (0);
5900182     }
5900183     //
5900184     // Otherwise, continue the recursion.
5900185     //
5900186     status = mkdir_parents (path_parent, mode);
5900187     if (status != 0)
5900188     {
5900189         return (-1);
5900190     }
5900191     //
5900192     // Previous directories are there: create the
5900193     // current one.
5900194     //
5900195     status = mkdir (path, mode);
5900196     if (status)
5900197     {
5900198         perror (path);
5900199         return (-1);
5900200     }
5900201
5900202     return (0);
5900203 }
5900204
5900205 -----
5900206 static void
5900207 usage (void)
5900208 {
5900209     fprintf
5900210         (stderr, "Usage: mkdir [-p] [-m OCTAL_MODE] DIR...\n");
5900211 }

```

## 96.1.26 applic/mmcheck.c

Si veda la sezione 86.18.

```

5910001 #include <sys/os32.h>
5910002 #include <kernel/memory.h>
5910003 #include <kernel/proc.h>
5910004 #include <unistd.h>
5910005 #include <stdio.h>
5910006 #include <fcntl.h>
5910007 #include <unistd.h>
5910008 #include <stdlib.h>
5910009 //-----
5910010 uint32_t mb_table[MEM_MAX_BLOCKS / 32]; // Memory
5910011 // blocks map.
5910012 unsigned int mb_max = MEM_MAX_BLOCKS; // Memory
5910013 // blocks max.
5910014 proc_t process;
5910015 //-----
5910016 static int mb_block_set0 (int block);
5910017 static void mb_check (pid_t pid, addr_t address,
5910018 size_t size);
5910019 static void mb_residual (void);
5910020 //-----
5910021 int
5910022 main (int argc, char *argv[], char *envp[])
5910023 {
5910024     int i;
5910025     int fd;
5910026     ssize_t size_read;
5910027     char *buffer;
5910028     pid_t pid;
5910029     proc_t *ps;
5910030     //
5910031     // Get memory map.
5910032     //
5910033     fd = open ("/dev/kmem_map", O_RDONLY);
5910034     if (fd < 0)
5910035     {
5910036         printf ("%s] Cannot open \"/dev/kmem_map\" ",
5910037             argv[0]);
5910038         perror (NULL);
5910039         return (0);
5910040     }
5910041     //
5910042     buffer = (char *) mb_table;
5910043     lseek (fd, (off_t) 0, SEEK_SET);
5910044     for (i = 0; i < (MEM_MAX_BLOCKS / 8); i += size_read)
5910045     {
5910046         size_read = read (fd, &buffer[i], BUFSIZ);
5910047         if (size_read < 0)
5910048         {
5910049             printf
5910050             ("%s] Cannot read "
5910051             "\"/dev/kmem_map\" %i %i ",
5910052             argv[0], size_read, sizeof (mb_table));
5910053             perror (NULL);
5910054             return (0);
5910055         }
5910056     }
5910057     //
5910058     close (fd);
5910059     //
5910060     // Scan processes
5910061     //
5910062     buffer = (char *) &process;
5910063     //
5910064     fd = open ("/dev/kmem_ps", O_RDONLY);
5910065     if (fd < 0)
5910066     {
5910067         printf ("%s] Cannot open \"/dev/kmem_ps\" ",
5910068             argv[0]);
5910069         perror (NULL);
5910070         exit (0);
5910071     }
5910072     //
5910073     // Scan processes.
5910074     //
5910075     for (pid = 0; pid < PROCESS_MAX; pid++)
5910076     {
5910077         lseek (fd, (off_t) pid, SEEK_SET);
5910078         size_read = read (fd, buffer, sizeof (proc_t));
5910079         if (size_read < sizeof (proc_t))
5910080         {
5910081             printf
5910082             ("%s] Cannot read "
5910083             "\"/dev/kmem_ps\" pid %i ", argv[0], pid);
5910084             perror (NULL);
5910085             continue;

```

```

5910086     }
5910087     ps = (proc_t *) buffer;
5910088     if (ps->status > 0)
5910089     {
5910090         //
5910091         //
5910092         //
5910093         if (ps->domain_data == 0)
5910094         {
5910095             mb_check (pid, ps->address_text,
5910096                 ps->domain_text + ps->extra_data);
5910097         }
5910098         else
5910099         {
5910100             mb_check (pid, ps->address_text,
5910101                 ps->domain_text);
5910102             mb_check (pid, ps->address_data,
5910103                 ps->domain_data + ps->extra_data);
5910104         }
5910105     }
5910106     }
5910107     close (fd);
5910108     //
5910109     // Check residual allocation, if any.
5910110     //
5910111     mb_residual ();
5910112     //
5910113     return (0);
5910114 }
5910115 //-----
5910116 static void
5910117 mb_check (pid_t pid, addr_t address, size_t size)
5910118 {
5910119     unsigned int bstart;
5910120     unsigned int bsize;
5910121     unsigned int bend;
5910122     unsigned int i;
5910123     addr_t block_address;
5910124     //
5910125     // k_printf ("releasing 0x%x, size 0x%x\n", (int)
5910126     // address,
5910127     // (int) size);
5910128     //
5910129     //
5910130     if (size == 0)
5910131     {
5910132         //
5910133         // Zero means nothing.
5910134         //
5910135         return;
5910136     }
5910137     //
5910138     if (size % MEM_BLOCK_SIZE)
5910139     {
5910140         bsize = size / MEM_BLOCK_SIZE + 1;
5910141     }
5910142     else
5910143     {
5910144         bsize = size / MEM_BLOCK_SIZE;
5910145     }
5910146     //
5910147     bstart = address / MEM_BLOCK_SIZE;
5910148     bend = bstart + bsize;
5910149     //
5910150     //
5910151     //
5910152     for (i = bstart; i < bend; i++)
5910153     {
5910154         if (mb_block_set0 (i))
5910155         {
5910156             ;
5910157         }
5910158         else
5910159         {
5910160             block_address = i;
5910161             block_address *= MEM_BLOCK_SIZE;
5910162             printf
5910163             ("PID %i is using memory address 0x%x, "
5910164             "but it is reported free or "
5910165             "already used by "
5910166             "another process!\n", (int) pid,
5910167             (unsigned int) block_address);
5910168         }
5910169     }
5910170     }
5910171 //-----
5910172

```



```

590173 static int
590174 mb_block_set0 (int block)
590175 {
590176     int i = block / 32;
590177     int j = block % 32;
590178     uint32_t mask = 0x80000000 >> j;
590179     if (mb_table[i] & mask)
590180     {
590181         mb_table[i] = mb_table[i] & ~mask;
590182         return (1);
590183     }
590184     else
590185     {
590186         return (0); // The block is already set to
590187                     // 0 inside the map!
590188     }
590189 }
590190
590191 //-----
590192 static void
590193 mb_residual (void)
590194 {
590195     unsigned int block;
590196     unsigned int blocks = MEM_MAX_BLOCKS;
590197     int i;
590198     int j;
590199     uint32_t mask;
590200     unsigned int start = 0;
590201     unsigned int stop = 0;
590202     unsigned int status = 0;
590203     //
590204     // Show residual allocated memory.
590205     //
590206     for (block = 0; block < blocks; block++)
590207     {
590208         i = block / 32;
590209         j = block % 32;
590210         mask = 0x80000000 >> j;
590211         if (mb_table[i] & mask)
590212         {
590213             //
590214             // Allocated block
590215             //
590216             if (status == 0)
590217             {
590218                 status = 1;
590219                 start = block;
590220             }
590221         }
590222         else
590223         {
590224             //
590225             // Not allocated block.
590226             //
590227             if (status == 1)
590228             {
590229                 status = 0;
590230                 stop = block;
590231             }
590232         }
590233         //
590234         //
590235         //
590236         if (stop > 0)
590237         {
590238             printf ("residual allocation: %x-%x ",
590239                 start, stop);
590240             start = 0;
590241             stop = 0;
590242         }
590243     }
590244     printf ("\n");
590245 }

```

## 96.1.27 applic/more.c

« Si veda la sezione 86.19.

```

592001 #include <unistd.h>
592002 #include <errno.h>
592003 //-----
592004 #define MAX_LINES 20
592005 #define MAX_COLUMNS 80
592006 //-----
592007 static void usage (void);
592008 //-----
592009 int

```

```

592010 main (int argc, char *argv[], char *envp[])
592011 {
592012     FILE *fp;
592013     char *name;
592014     int c;
592015     int line = 1; // Line internal counter.
592016     int column = 1; // Column internal counter.
592017     int a; // Index inside arguments.
592018     int loop;
592019     //
592020     // There must be at least an argument, plus the
592021     // program name.
592022     //
592023     if (argc < 2)
592024     {
592025         usage ();
592026         return (1);
592027     }
592028     //
592029     // No options are allowed.
592030     //
592031     for (a = 1; a < argc; a++)
592032     {
592033         //
592034         // Get next name from arguments.
592035         //
592036         name = argv[a];
592037         //
592038         // Try to open the file, read only.
592039         //
592040         fp = fopen (name, "r");
592041         //
592042         if (fp == NULL)
592043         {
592044             //
592045             // Error opening file.
592046             //
592047             perror (name);
592048             return (1);
592049         }
592050         //
592051         // Print the file name to be displayed.
592052         //
592053         printf ("== %s ==\n", name);
592054         line++;
592055         //
592056         // The following loop continues while the file
592057         // gives characters, or when a command to change
592058         // file or to quit is given.
592059         //
592060         for (loop = 1; loop;)
592061         {
592062             //
592063             // Read a single character.
592064             //
592065             c = getc (fp);
592066             //
592067             if (c == EOF)
592068             {
592069                 loop = 0;
592070                 break;
592071             }
592072             //
592073             // If the character read is a special one,
592074             // the line/column calculation is modified,
592075             // so that it is known when to stop
592076             // scrolling.
592077             //
592078             switch (c)
592079             {
592080                 case '\r':
592081                     //
592082                     // Displaying this character, the cursor
592083                     // should go
592084                     // back to the first column. So the
592085                     // column counter
592086                     // is reset.
592087                     //
592088                     column = 1;
592089                     break;
592090                 case '\n':
592091                     //
592092                     // Displaying this character, the cursor
592093                     // should go
592094                     // back to the next line, at the first
592095                     // column.
592096                     // So the column counter is reset and

```



```

5930035     if (strcmp (argv[3], "rw") == 0)
5930036     {
5930037         options = MOUNT_DEFAULT;
5930038     }
5930039     else if (strcmp (argv[3], "ro") == 0)
5930040     {
5930041         options = MOUNT_RO;
5930042     }
5930043     else
5930044     {
5930045         printf
5930046         ("Invalid mount option: "
5930047          "only \"ro\" or \"rw\" " "are allowed\n");
5930048         return (2);
5930049     }
5930050 }
5930051 else
5930052 {
5930053     options = MOUNT_DEFAULT;
5930054 }
5930055 //
5930056 // System call.
5930057 //
5930058 status = mount (argv[1], argv[2], options);
5930059 if (status != 0)
5930060 {
5930061     perror (NULL);
5930062     return (2);
5930063 }
5930064 //
5930065 return (0);
5930066 }
5930067
5930068 -----
5930069 static void
5930070 usage (void)
5930071 {
5930072     fprintf (stderr, "Usage: mount DEVICE MOUNT_POINT "
5930073             "[MOUNT_OPTIONS]\n");
5930074 }

```

### 96.1.29 applic/nc.c

« Si veda la sezione 86.20.

```

5940001 #include <sys/stat.h>
5940002 #include <sys/types.h>
5940003 #include <unistd.h>
5940004 #include <stdlib.h>
5940005 #include <fcntl.h>
5940006 #include <errno.h>
5940007 #include <signal.h>
5940008 #include <stdio.h>
5940009 #include <string.h>
5940010 #include <limits.h>
5940011 #include <libgen.h>
5940012 #include <arpa/inet.h>
5940013 #include <sys/socket.h>
5940014 #include <stdint.h>
5940015 #include <stdbool.h>
5940016 #include <fcntl.h>
5940017 -----
5940018 static void usage (void);
5940019 char buffer[BUFSIZ];
5940020 -----
5940021 int
5940022 main (int argc, char *argv[], char *envp[])
5940023 {
5940024     bool option_l = 0;
5940025     bool option_u = 0;
5940026     int opt;
5940027 //extern char *optarg; // not used.
5940028 extern int optind;
5940029 extern int optopt;
5940030 //
5940031 int status;
5940032 int sfdn;
5940033 int sfdn2;
5940034 struct sockaddr_in sa_local;
5940035 struct sockaddr_in sa_remote;
5940036 socklen_t sa_remote_size = sizeof (struct sockaddr_in);
5940037 ssize_t read_size;
5940038 ssize_t sent_size;
5940039 ssize_t recv_size;
5940040 char *addr = NULL;
5940041 char *port = NULL;
5940042 bool can_rx = 1;

```

```

5940043 bool can_tx = 1;
5940044 //
5940045 // Check for options.
5940046 //
5940047 while ((opt = getopt (argc, argv, ":ul")) != -1)
5940048 {
5940049     switch (opt)
5940050     {
5940051         case 'l':
5940052             option_l = 1;
5940053             break;
5940054         case 'u':
5940055             option_u = 1;
5940056             break;
5940057         case '?':
5940058             fprintf (stderr, "Unknown option -%c.\n", optopt);
5940059             usage ();
5940060             return (1);
5940061             break;
5940062         case ':':
5940063             fprintf (stderr,
5940064                     "Missing argument for option -%c\n",
5940065                     optopt);
5940066             usage ();
5940067             return (1);
5940068             break;
5940069         default:
5940070             fprintf (stderr,
5940071                     "Getopt problem: "
5940072                     "unknown option %c\n", opt);
5940073             usage ();
5940074             return (1);
5940075     }
5940076 }
5940077 //
5940078 // Arguments.
5940079 //
5940080 if (optind == (argc - 2))
5940081 {
5940082     //
5940083     // There are exactly two arguments: destination
5940084     // address and port.
5940085     //
5940086     addr = argv[argc - 2];
5940087     port = argv[argc - 1];
5940088 }
5940089 else
5940090 {
5940091     //
5940092     // Arguments wrong!
5940093     //
5940094     usage ();
5940095     return (2);
5940096 }
5940097 //
5940098 // Set the local or the remote address.
5940099 //
5940100 if (option_l)
5940101 {
5940102     //
5940103     // Address and port are local.
5940104     //
5940105     sa_local.sin_family = AF_INET;
5940106     sa_local.sin_port = htons (atoi (port));
5940107     inet_pton (AF_INET, addr, &sa_local.sin_addr.s_addr);
5940108 }
5940109 else
5940110 {
5940111     //
5940112     // Address and port are remote.
5940113     //
5940114     sa_remote.sin_family = AF_INET;
5940115     sa_remote.sin_port = htons (atoi (port));
5940116     inet_pton (AF_INET, addr, &sa_remote.sin_addr.s_addr);
5940117 }
5940118 //
5940119 // Open the socket.
5940120 //
5940121 if (option_u)
5940122 {
5940123     sfdn = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
5940124 }
5940125 else
5940126 {
5940127     sfdn = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
5940128 }
5940129 if (sfdn < 0)

```

```

5940130 {
5940131     perror (NULL);
5940132     return (3);
5940133 }
5940134 //
5940135 // Set it listening or connect.
5940136 //
5940137 if (option_1)
5940138 {
5940139     //
5940140     // Bind the local 'sa' location.
5940141     //
5940142     status =
5940143         bind (sfdn, (struct sockaddr *) &sa_local,
5940144              sizeof (sa_local));
5940145     if (status < 0)
5940146     {
5940147         perror (NULL);
5940148         close (sfdn);
5940149         return (4);
5940150     }
5940151     //
5940152     // Listen (TCP) or wait the first packet (UDP).
5940153     //
5940154     if (option_u)
5940155     {
5940156         //
5940157         // Instead of listening, we use the function
5940158         // 'recvfrom()',
5940159         // to get the remote address and port.
5940160         //
5940161         recv_size =
5940162             recvfrom (sfdn, &buffer,
5940163                      (size_t) BUFSIZ - 1, 0,
5940164                      (struct sockaddr *) &sa_remote,
5940165                      &sa_remote_size);
5940166         if (recv_size < 0)
5940167         {
5940168             perror (NULL);
5940169             close (sfdn);
5940170             return (4);
5940171         }
5940172         //
5940173         // Now connect the remote destination
5940174         //
5940175         status =
5940176             connect (sfdn,
5940177                     (struct sockaddr *) &sa_remote,
5940178                     sizeof (sa_remote));
5940179         if (status < 0)
5940180         {
5940181             perror (NULL);
5940182             close (sfdn);
5940183             return (7);
5940184         }
5940185         //
5940186         // And show what was received as a first
5940187         // packet.
5940188         //
5940189         buffer[recv_size] = 0;
5940190         printf ("%s", buffer);
5940191     }
5940192     else
5940193     {
5940194         //
5940195         // TCP: listen.
5940196         //
5940197         status = listen (sfdn, 1);
5940198         if (status < 0)
5940199         {
5940200             perror (NULL);
5940201             close (sfdn);
5940202             return (5);
5940203         }
5940204         //
5940205         // Accept.
5940206         //
5940207         sfdn2 =
5940208             accept (sfdn,
5940209                    (struct sockaddr *) &sa_remote,
5940210                    &sa_remote_size);
5940211         if (sfdn2 < 0)
5940212         {
5940213             perror (NULL);
5940214             close (sfdn);
5940215             return (6);
5940216         }

```

```

5940217     //
5940218     // Close listening socket.
5940219     //
5940220     close (sfdn);
5940221     //
5940222     // Variable 'sfdn' will be the new socket.
5940223     //
5940224     sfdn = sfdn2;
5940225     }
5940226 }
5940227 else
5940228 {
5940229     //
5940230     // Connect the remote destination.
5940231     //
5940232     status =
5940233         connect (sfdn, (struct sockaddr *) &sa_remote,
5940234                 sizeof (sa_remote));
5940235     if (status < 0)
5940236     {
5940237         perror (NULL);
5940238         close (sfdn);
5940239         return (7);
5940240     }
5940241     }
5940242     //
5940243     // Define the standard input non blocking.
5940244     //
5940245     status = fcntl (STDIN_FILENO, F_SETFL, O_NONBLOCK);
5940246     if (status < 0)
5940247     {
5940248         perror (NULL);
5940249         return (8);
5940250     }
5940251     //
5940252     // Define the socket non blocking.
5940253     //
5940254     status = fcntl (sfdn, F_SETFL, O_NONBLOCK);
5940255     if (status < 0)
5940256     {
5940257         perror (NULL);
5940258         return (9);
5940259     }
5940260     //
5940261     // Will read from the remote and show to the screen.
5940262     //
5940263     while (can_rx || can_tx)
5940264     {
5940265         if (can_rx)
5940266         {
5940267             recv_size =
5940268                 recv (sfdn, &buffer, (size_t) BUFSIZ - 1, 0);
5940269             //
5940270             // recv_size = read (sfdn, &buffer,
5940271                               // (size_t) BUFSIZ-1);
5940272             if (recv_size < 0)
5940273             {
5940274                 if (errno == EAGAIN || errno == EWOULDBLOCK)
5940275                 {
5940276                     ;
5940277                 }
5940278                 else
5940279                 {
5940280                     perror (NULL);
5940281                     close (sfdn);
5940282                     return (10);
5940283                 }
5940284             }
5940285             else if (recv_size == 0)
5940286             {
5940287                 //
5940288                 // End of stream.
5940289                 //
5940290                 can_rx = 0;
5940291                 printf ("--end of receive stream--\n");
5940292             }
5940293             else
5940294             {
5940295                 buffer[recv_size] = 0;
5940296                 printf ("%s", buffer);
5940297             }
5940298         }
5940299         if (can_tx)
5940300         {
5940301             read_size = read (STDIN_FILENO, buffer, BUFSIZ);
5940302             if (read_size < 0)
5940303             {
5940304                 if (errno == EAGAIN || errno == EWOULDBLOCK)

```

```

5940304     {
5940305     ;
5940306     }
5940307     else
5940308     {
5940309         perror (NULL);
5940310         close (sfdn);
5940311         return (11);
5940312     }
5940313     }
5940314     else if (read_size == 0)
5940315     {
5940316         //
5940317         // End of input.
5940318         //
5940319         printf ("--closing send stream--\n");
5940320         can_tx = 0;
5940321     }
5940322     else
5940323     {
5940324         //
5940325         // Send it.
5940326         //
5940327         sent_size =
5940328         send (sfdn, &buffer, (size_t) read_size, 0);
5940329         if (sent_size < 0)
5940330         {
5940331             if (errno == EAGAIN
5940332                 || errno == EWOULDBLOCK)
5940333             {
5940334                 ;
5940335             }
5940336             else
5940337             {
5940338                 perror (NULL);
5940339                 close (sfdn);
5940340                 return (12);
5940341             }
5940342         }
5940343     }
5940344     }
5940345     }
5940346     //
5940347     // All done.
5940348     //
5940349     close (sfdn);
5940350     return (0);
5940351 }
5940352
5940353 //-----
5940354 static void
5940355 usage (void)
5940356 {
5940357     fprintf
5940358     (stderr,
5940359      "os32 netcat usage:\n"
5940360      "\n"
5940361      "nc [-u][-l] ADDRESS PORT\n"
5940362      "\n"
5940363      "-u      Use UDP protocol instead of TCP.\n"
5940364      "-l      Listen for incoming connection \n"
5940365      "requests.\n"
5940366      "ADDRESS IPv4 numeric address; if option -l is\n"
5940367      "used, this\n"
5940368      "is the local address, otherwise it is\n"
5940369      "the remote address.\n"
5940370      "PORT   TCP or UDP port; if option -l is used,\n"
5940371      "this is local address, otherwise it is\n"
5940372      "the remote address.\n");
5940373 }

```

### 96.1.30 applic/ping.c

« Si veda la sezione 92.8.

```

5950001 #include <stdio.h>
5950002 #include <sys/types.h>
5950003 #include <arpa/inet.h>
5950004 #include <sys/socket.h>
5950005 #include <netinet/icmp.h>
5950006 #include <unistd.h>
5950007 #include <stdlib.h>
5950008 #include <stdint.h>
5950009 #include <errno.h>
5950010 //-----
5950011 struct ip_pkt
5950012 {

```

```

5950013 struct iphdr ip; // <netinet/ip.h>
5950014 struct icmp_hdr icmp; // <netinet/icmp.h>
5950015 char data[60];
5950016 } __attribute__ ((packed));
5950017
5950018 struct icmp_pkt
5950019 {
5950020     struct icmp_hdr icmp; // <netinet/icmp.h>
5950021     char data[60];
5950022 } __attribute__ ((packed));
5950023 //-----
5950024 static uint16_t ip_chk (uint16_t * data, size_t size);
5950025 static void usage (void);
5950026 //-----
5950027 int
5950028 main (int argc, char *argv[], char *envp[])
5950029 {
5950030     int sfdn;
5950031     struct sockaddr_in sa;
5950032     ssize_t sent;
5950033     ssize_t received;
5950034     int status;
5950035     char *destination;
5950036     uint16_t checksum;
5950037     struct icmp_pkt icmp_pkt_send;
5950038     struct ip_pkt ip_pkt_receive;
5950039     //int      retry = 3; // Max send retry.
5950040     clock_t clock_ping;
5950041     clock_t clock_pong;
5950042     clock_t clock_time;
5950043     //
5950044     // No options are known, but an argument must be
5950045     // given.
5950046     //
5950047     if (argc < 2)
5950048     {
5950049         usage ();
5950050         return (1);
5950051     }
5950052     destination = argv[1];
5950053     //
5950054     // Define the destination 'sa'
5950055     //
5950056     sa.sin_family = AF_INET;
5950057     sa.sin_port = 0;
5950058     inet_pton (AF_INET, destination, &sa.sin_addr.s_addr);
5950059     //
5950060     // Put some data inside the packet header.
5950061     //
5950062     icmp_pkt_send.icmp.un.echo.id = (uint16_t) rand ();
5950063     icmp_pkt_send.icmp.un.echo.sequence = 0;
5950064     icmp_pkt_send.icmp.type = 8; // Echo request.
5950065     icmp_pkt_send.icmp.code = 0;
5950066     icmp_pkt_send.icmp.checksum = 0;
5950067     //
5950068     // Calculate the ICMP checksum.
5950069     //
5950070     checksum = ~(ip_chk ((void *) &icmp_pkt_send,
5950071                          sizeof (struct icmp_pkt)));
5950072     icmp_pkt_send.icmp.checksum = htons (checksum);
5950073     //
5950074     // Open the socket.
5950075     //
5950076     sfdn = socket (AF_INET, SOCK_RAW, IPPROTO_ICMP);
5950077     if (sfdn < 0)
5950078     {
5950079         perror (NULL);
5950080         return (0);
5950081     }
5950082     //
5950083     // Connect the 'sa' destination
5950084     //
5950085     status =
5950086     connect (sfdn, (struct sockaddr *) &sa, sizeof (sa));
5950087     if (status < 0)
5950088     {
5950089         perror (NULL);
5950090         close (sfdn);
5950091         return (0);
5950092     }
5950093     //
5950094     // Send one single packet: please notice that we
5950095     // send an ICMP,
5950096     // but we receive a full IP.
5950097     //
5950098     clock_ping = clock ();
5950099     sent = send (sfdn, &icmp_pkt_send,

```

```

3990100         sizeof (struct icmp_pkt), 0);
3990101     if (sent < 0)
3990102     {
3990103         perror (NULL);
3990104         return (1);
3990105     }
3990106     //
3990107     // Packet sent.
3990108     //
3990109     printf ("ping: ");
3990110     //
3990111     // Now receive all ICMP raw packets, and select the
3990112     // one with the same
3990113     // identifier.
3990114     //
3990115     while (1)
3990116     {
3990117         received =
3990118             read (sfdn, &ip_pkt_receive,
3990119                 sizeof (struct ip_pkt));
3990120         clock_pong = clock ();
3990121
3990122         if (ip_pkt_receive.icmp.un.echo.id
3990123             == icmp_pkt_send.icmp.un.echo.id)
3990124         {
3990125             clock_time = (clock_pong - clock_ping);
3990126             printf ("pong %llu.%03llu s\n",
3990127                 clock_time / CLOCKS_PER_SEC,
3990128                 (clock_time % CLOCKS_PER_SEC) *
3990129                 1000 / CLOCKS_PER_SEC);
3990130             break;
3990131         }
3990132     }
3990133
3990134     close (sfdn);
3990135     return (0);
3990136 }
3990137
3990138 //-----
3990139 static uint16_t
3990140 ip_chk (uint16_t * data, size_t size)
3990141 {
3990142     int i;
3990143     uint32_t sum;
3990144     uint16_t carry;
3990145     uint16_t checksum;
3990146     uint16_t last;
3990147     uint8_t *octet = (uint8_t *) data;
3990148     //
3990149     // 2's complement sum.
3990150     //
3990151     for (i = 0, sum = 0; i < (size / 2); i++)
3990152     {
3990153         sum += ntohs (data[i]);
3990154     }
3990155     //
3990156     if (size % 2)
3990157     {
3990158         //
3990159         // The size is odd, and the last octet must be
3990160         // accounted too.
3990161         //
3990162         last = octet[size - 1];
3990163         last = last << 8;
3990164         //
3990165         sum += last;
3990166     }
3990167     //
3990168     // Extract the carries and make the checksum.
3990169     //
3990170     carry = sum >> 16;
3990171     checksum = sum & 0x0000FFFF;
3990172     checksum += carry;
3990173     //
3990174     // End of job.
3990175     //
3990176     return (checksum);
3990177 }
3990178
3990179 //-----
3990180 static void
3990181 usage (void)
3990182 {
3990183     fprintf (stderr, "Usage: ping IPv4\n");
3990184 }

```

## 96.1.31 applic/ps.c

Si veda la sezione 86.21.

```

5960001 #include <sys/os32.h>
5960002 #include <kernel/proc.h>
5960003 #include <unistd.h>
5960004 #include <stdio.h>
5960005 #include <fcntl.h>
5960006 #include <unistd.h>
5960007 #include <stdlib.h>
5960008 //-----
5960009 static void usage (void);
5960010 //-----
5960011 int
5960012 main (int argc, char *argv[], char *envp[])
5960013 {
5960014     pid_t pid;
5960015     proc_t *ps;
5960016     int fd;
5960017     char stat;
5960018     ssize_t size_read;
5960019     char buffer[sizeof (proc_t)];
5960020     unsigned int min;
5960021     unsigned int sec;
5960022     size_t stack_size;
5960023     addr_t stack_bottom;
5960024     int stack_usage;
5960025     //
5960026     int opt;
5960027     // extern char *optarg;           // not used.
5960028     // extern int optind;
5960029     extern int optopt;
5960030     int option_u = 0;
5960031     int option_g = 0;
5960032     //
5960033     // Check for options.
5960034     //
5960035     while ((opt = getopt (argc, argv, "ug")) != -1)
5960036     {
5960037         switch (opt)
5960038         {
5960039             case 'u':
5960040                 option_u = 1;
5960041                 break;
5960042             case 'g':
5960043                 option_g = 1;
5960044                 break;
5960045             case '?':
5960046                 fprintf (stderr, "Unknown option -%c.\n", optopt);
5960047                 usage ();
5960048                 return (1);
5960049             case ':':
5960050                 fprintf (stderr,
5960051                     "Missing argument for option -%c\n",
5960052                     optopt);
5960053                 usage ();
5960054                 return (1);
5960055             case ':':
5960056                 break;
5960057             default:
5960058                 fprintf (stderr,
5960059                     "Getopt problem: "
5960060                     "unknown option %c\n", opt);
5960061                 return (1);
5960062         }
5960063     }
5960064     //
5960065     // Fix options '-u' or '-g'.
5960066     //
5960067     if (option_u)
5960068         option_g = 0;
5960069     if (!option_g)
5960070         option_u = 1;
5960071     //
5960072     // Open '/dev/kmem_ps', to get the process table.
5960073     //
5960074     fd = open ("/dev/kmem_ps", O_RDONLY);
5960075     if (fd < 0)
5960076     {
5960077         printf ("[%s] Cannot open \"/dev/kmem_ps\" ",
5960078             argv[0]);
5960079         perror (NULL);
5960080         exit (0);
5960081     }
5960082     //
5960083     // Print head.
5960084     //
5960085     if (option_u)

```

```

5960086 {
5960087     printf ("pp p pg                "
5960088            "T * 0x1000 D * 0x1000 stack      \n"
5960089            "id id rp tty uid euid suid usage s "
5960090            "addr size addr size usage  name\n");
5960091 }
5960092 else
5960093 {
5960094     printf ("pp p pg                "
5960095            "T * 0x1000 D * 0x1000 stack      \n"
5960096            "id id rp tty gid egid sgid usage s "
5960097            "addr size addr size usage  name\n");
5960098 }
5960099 //
5960100 // Scan processes and then print body.
5960101 //
5960102 for (pid = 0; pid < PROCESS_MAX; pid++)
5960103 {
5960104     lseek (fd, (off_t) pid, SEEK_SET);
5960105     size_read = read (fd, buffer, sizeof (proc_t));
5960106     if (size_read < sizeof (proc_t))
5960107     {
5960108         printf
5960109             ("[%s] Cannot read "
5960110              "\"/dev/kmem_ps\" pid %i ", argv[0], pid);
5960111         perror (NULL);
5960112         continue;
5960113     }
5960114     ps = (proc_t *) buffer;
5960115     if (ps->status > 0)
5960116     {
5960117         ps->name[PATH_MAX - 1] = 0; // Terminated
5960118         // string.
5960119         //
5960120         // Check the current stack size.
5960121         //
5960122         if (ps->domain_data == 0)
5960123         {
5960124             stack_bottom = ps->domain_text;
5960125         }
5960126         else
5960127         {
5960128             stack_bottom = ps->domain_data;
5960129         }
5960130         //
5960131         stack_size = stack_bottom - ps->sp;
5960132         //
5960133         stack_usage = 100 * stack_size / ps->domain_stack;
5960134         //
5960135         switch (ps->status)
5960136         {
5960137             case PROC_EMPTY:
5960138                 stat = '-';
5960139                 break;
5960140             case PROC_CREATED:
5960141                 stat = 'c';
5960142                 break;
5960143             case PROC_READY:
5960144                 stat = 'r';
5960145                 break;
5960146             case PROC_RUNNING:
5960147                 stat = 'R';
5960148                 break;
5960149             case PROC_SLEEPING:
5960150                 stat = 's';
5960151                 break;
5960152             case PROC_ZOMBIE:
5960153                 stat = 'z';
5960154                 break;
5960155             default:
5960156                 stat = '?';
5960157                 break;
5960158         }
5960159         //
5960160         min = ((ps->usage / CLOCKS_PER_SEC) / 60);
5960161         sec = ((ps->usage / CLOCKS_PER_SEC) % 60);
5960162         //
5960163         // Print the line.
5960164         //
5960165         //
5960166         // Addresses and sizes are multiple of 4096
5960167         // (0x1000);
5960168         // For the stack pointer is shown only the
5960169         // last five
5960170         // hexadecimal digits.
5960171         //
5960172         if (ps->domain_data > 0)

```

```

5960173     {
5960174         if (option_u)
5960175         {
5960176             printf
5960177                 ("%2i %2i %2i %04x %4i %4i %4i "
5960178                  "%02i.%02i %c %05x %04x %05x "
5960179                  "%04x % 3i%% %15s\n",
5960180                  (unsigned int) ps->ppid,
5960181                  (unsigned int) pid,
5960182                  (unsigned int) ps->pgrp,
5960183                  (unsigned int) ps->device_tty,
5960184                  (unsigned int) ps->uid,
5960185                  (unsigned int) ps->euid,
5960186                  (unsigned int) ps->suid, min, sec,
5960187                  stat,
5960188                  (unsigned int) ps->address_text /
5960189                  MEM_BLOCK_SIZE,
5960190                  (unsigned int) ps->domain_text /
5960191                  MEM_BLOCK_SIZE,
5960192                  (unsigned int) ps->address_data /
5960193                  MEM_BLOCK_SIZE,
5960194                  (unsigned int) (ps->domain_data +
5960195                               ps->extra_data) /
5960196                  MEM_BLOCK_SIZE,
5960197                  (unsigned int) stack_usage, ps->name);
5960198         }
5960199     }
5960200     else
5960201     {
5960202         printf
5960203             ("%2i %2i %2i %04x %4i %4i "
5960204              "%4i %02i.%02i %c %05x %04x "
5960205              "%05x %04x % 3i%% %15s\n",
5960206              (unsigned int) ps->ppid,
5960207              (unsigned int) pid,
5960208              (unsigned int) ps->pgrp,
5960209              (unsigned int) ps->device_tty,
5960210              (unsigned int) ps->gid,
5960211              (unsigned int) ps->egid,
5960212              (unsigned int) ps->sgid, min, sec,
5960213              stat,
5960214              (unsigned int) ps->address_text /
5960215              MEM_BLOCK_SIZE,
5960216              (unsigned int) ps->domain_text /
5960217              MEM_BLOCK_SIZE,
5960218              (unsigned int) ps->address_data /
5960219              MEM_BLOCK_SIZE,
5960220              (unsigned int) (ps->domain_data +
5960221                           ps->extra_data) /
5960222              MEM_BLOCK_SIZE,
5960223              (unsigned int) stack_usage, ps->name);
5960224     }
5960225     }
5960226     else
5960227     {
5960228         if (option_u)
5960229         {
5960230             printf
5960231                 ("%2i %2i %2i %04x %4i %4i %4i "
5960232                  "%02i.%02i %c %05x %04x %05x "
5960233                  "%04x % 3i%% %15s\n",
5960234                  (unsigned int) ps->ppid,
5960235                  (unsigned int) pid,
5960236                  (unsigned int) ps->pgrp,
5960237                  (unsigned int) ps->device_tty,
5960238                  (unsigned int) ps->uid,
5960239                  (unsigned int) ps->euid,
5960240                  (unsigned int) ps->suid, min, sec,
5960241                  stat,
5960242                  (unsigned int) ps->address_text /
5960243                  MEM_BLOCK_SIZE,
5960244                  (unsigned int) (ps->domain_text +
5960245                               ps->extra_data) /
5960246                  MEM_BLOCK_SIZE,
5960247                  (unsigned int) ps->address_data /
5960248                  MEM_BLOCK_SIZE,
5960249                  (unsigned int) ps->domain_data /
5960250                  MEM_BLOCK_SIZE,
5960251                  (unsigned int) stack_usage, ps->name);
5960252         }
5960253     }
5960254     else
5960255     {
5960256         printf
5960257             ("%2i %2i %2i %04x %4i %4i %4i "
5960258              "%02i.%02i %c %05x %04x %05x "
5960259              "%04x % 3i%% %15s\n",
5960260              (unsigned int) ps->ppid,
5960261              (unsigned int) pid,

```

```

5960260 (unsigned int) ps->pgrp,
5960261 (unsigned int) ps->device_tty,
5960262 (unsigned int) ps->gid,
5960263 (unsigned int) ps->egid,
5960264 (unsigned int) ps->sgid, min, sec,
5960265 stat,
5960266 (unsigned int) ps->address_text /
5960267 MEM_BLOCK_SIZE,
5960268 (unsigned int) (ps->domain_text +
5960269 ps->extra_data) /
5960270 MEM_BLOCK_SIZE,
5960271 (unsigned int) ps->address_data /
5960272 MEM_BLOCK_SIZE,
5960273 (unsigned int) ps->domain_data /
5960274 MEM_BLOCK_SIZE,
5960275 (unsigned int) stack_usage, ps->name);
5960276 }
5960277 }
5960278 }
5960279 }
5960280 close (fd);
5960281 return (0);
5960282 }
5960283 }
5960284 //-----
5960285 static void
5960286 usage (void)
5960287 {
5960288     fprintf (stderr, "Usage: ps [-u|g]\n");
5960289 }

```

### 96.1.32 applic/rm.c

« Si veda la sezione 86.22.

```

5970001 #include <fcntl.h>
5970002 #include <sys/stat.h>
5970003 #include <stddef.h>
5970004 #include <unistd.h>
5970005 #include <errno.h>
5970006 //-----
5970007 static void usage (void);
5970008 //-----
5970009 int
5970010 main (int argc, char *argv[], char *envp[])
5970011 {
5970012     int a;          // Argument index.
5970013     int status;
5970014     struct stat file_status;
5970015     //
5970016     // No options are known, but at least an argument
5970017     // must be given.
5970018     //
5970019     if (argc < 2)
5970020     {
5970021         usage ();
5970022         return (1);
5970023     }
5970024     //
5970025     // Scan arguments.
5970026     //
5970027     for (a = 1; a < argc; a++)
5970028     {
5970029         //
5970030         // Verify if the file exists.
5970031         //
5970032         if (stat (argv[a], &file_status) != 0)
5970033         {
5970034             fprintf (stderr,
5970035                 "File \"%s\" does not exist!\n",
5970036                 argv[a]);
5970037             continue;
5970038         }
5970039         //
5970040         // File exists: check the file type.
5970041         //
5970042         if (S_ISDIR (file_status.st_mode))
5970043         {
5970044             fprintf (stderr,
5970045                 "Cannot remove directory \"%s\"!\n",
5970046                 argv[a]);
5970047             continue;
5970048         }
5970049         //
5970050         // Can remove it.
5970051         //
5970052         status = unlink (argv[a]);

```

```

5970053     if (status != 0)
5970054     {
5970055         perror (NULL);
5970056         return (2);
5970057     }
5970058 }
5970059 return (0);
5970060 }
5970061 //-----
5970062 static void
5970063 usage (void)
5970064 {
5970065     fprintf (stderr, "Usage: rm FILE...\n");
5970066 }
5970067 }

```

### 96.1.33 applic/rmdir.c

« Si veda la sezione 86.23.

```

5980001 #include <fcntl.h>
5980002 #include <sys/stat.h>
5980003 #include <stddef.h>
5980004 #include <unistd.h>
5980005 #include <errno.h>
5980006 //-----
5980007 static void usage (void);
5980008 //-----
5980009 int
5980010 main (int argc, char *argv[], char *envp[])
5980011 {
5980012     int a;          // Argument index.
5980013     int status;
5980014     struct stat file_status;
5980015     //
5980016     // No options are known, but at least an argument
5980017     // must be given.
5980018     //
5980019     if (argc < 2)
5980020     {
5980021         usage ();
5980022         return (1);
5980023     }
5980024     //
5980025     // Scan arguments.
5980026     //
5980027     for (a = 1; a < argc; a++)
5980028     {
5980029         //
5980030         // Verify if the file exists.
5980031         //
5980032         if (stat (argv[a], &file_status) != 0)
5980033         {
5980034             fprintf (stderr,
5980035                 "File \"%s\" does not exist!\n",
5980036                 argv[a]);
5980037             continue;
5980038         }
5980039         //
5980040         // File exists: check the file type.
5980041         //
5980042         if (!S_ISDIR (file_status.st_mode))
5980043         {
5980044             fprintf (stderr,
5980045                 "Cannot remove file \"%s\"!\n",
5980046                 argv[a]);
5980047             continue;
5980048         }
5980049         //
5980050         // Can try to remove it.
5980051         //
5980052         status = rmdir (argv[a]);
5980053         if (status != 0)
5980054         {
5980055             perror (NULL);
5980056             return (2);
5980057         }
5980058     }
5980059 return (0);
5980060 }
5980061 //-----
5980062 static void
5980063 usage (void)
5980064 {
5980065     fprintf (stderr, "Usage: rmdir DIR...\n");
5980066 }

```



## 96.1.34 applic/route.c

« Si veda la sezione 92.9.

```

599001 #include <sys/os32.h>
599002 #include <kernel/net/route.h>
599003 #include <kernel/net.h>
599004 #include <unistd.h>
599005 #include <stdio.h>
599006 #include <fcntl.h>
599007 #include <unistd.h>
599008 #include <stdlib.h>
599009 //-----
599010 int
599011 main (int argc, char *argv[], char *envp[])
599012 {
599013     int fd;
599014     ssize_t size_read;
599015     char buffer[sizeof (route_t)];
599016     int n;
599017     route_t *route_table_item;
599018     char string[80];
599019
599020     //
599021     // All options are ignored, at the moment
599022     //
599023
599024     //
599025     // Open '/dev/kmem_route', to get the routing table.
599026     //
599027     fd = open ("/dev/kmem_route", O_RDONLY);
599028     if (fd < 0)
599029     {
599030         printf ("%s] Cannot open '/dev/kmem_route' ",
599031                argv[0]);
599032         perror (NULL);
599033         exit (0);
599034     }
599035     //
599036     // Print header.
599037     //
599038     printf ("Destination/mask      "
599039            "Router                " "Interface\n");
599040     //
599041     // Scan route table items and then print body.
599042     //
599043     for (n = 0; n < ROUTE_MAX_ROUTES; n++)
599044     {
599045         lseek (fd, (off_t) n, SEEK_SET);
599046         size_read = read (fd, buffer, sizeof (route_t));
599047         if (size_read < sizeof (route_t))
599048         {
599049             printf
599050                 ("%s] Cannot read "
599051                  "%s]/dev/kmem_route" item %i ", argv[0], n);
599052             perror (NULL);
599053             continue;
599054         }
599055         //
599056         route_table_item = (route_t *) buffer;
599057         //
599058         if (route_table_item->network == 0xFFFFFFFF)
599059         {
599060             //
599061             // Empty item.
599062             //
599063             continue;
599064         }
599065         //
599066         sprintf (string, "%i.%i.%i.%i/%i"
599067                 "          ",
599068                 route_table_item->network >> 24 & 0x000000FF,
599069                 route_table_item->network >> 16 & 0x000000FF,
599070                 route_table_item->network >> 8 & 0x000000FF,
599071                 route_table_item->network >> 0 & 0x000000FF,
599072                 route_table_item->m);
599073         string[19] = '\0';
599074         printf ("%s", string);
599075         //
599076         if (route_table_item->router == 0)
599077         {
599078             printf ("          ");
599079         }
599080         else
599081         {
599082             sprintf (string, "%i.%i.%i.%i"
599083                     "          ",
599084                     route_table_item->router >> 24 &
599085                     0x000000FF,

```

```

599086         route_table_item->router >> 16 &
599087         0x000000FF,
599088         route_table_item->router >> 8 &
599089         0x000000FF,
599090         route_table_item->router >> 0 &
599091         0x000000FF);
599092         string[16] = '\0';
599093         printf ("%s", string);
599094     }
599095     //
599096     printf ("net%i\n", route_table_item->interface);
599097 }
599098 close (fd);
599099 return (0);
599100 }

```

## 96.1.35 applic/shell.c

« Si veda la sezione 86.24.

```

600001 #include <unistd.h>
600002 #include <stdlib.h>
600003 #include <sys/stat.h>
600004 #include <sys/types.h>
600005 #include <fcntl.h>
600006 #include <errno.h>
600007 #include <unistd.h>
600008 #include <signal.h>
600009 #include <stdio.h>
600010 #include <sys/wait.h>
600011 #include <stdio.h>
600012 #include <string.h>
600013 #include <limits.h>
600014 #include <sys/os32.h>
600015 //-----
600016 #define PROMPT_SIZE      16
600017 //-----
600018 static void sh_cd (int argc, char *argv[]);
600019 static void sh_pwd (int argc, char *argv[]);
600020 static void sh_umask (int argc, char *argv[]);
600021 //-----
600022 int
600023 main (int argc, char *argv[], char *envp[])
600024 {
600025     char buffer_cmd[ARG_MAX / 2];
600026     char *argv_cmd[ARG_MAX / 16];
600027     //char prompt[PROMPT_SIZE];
600028     uid_t uid;
600029     int argc_cmd;
600030     pid_t pid_cmd;
600031     pid_t pid_dead;
600032     int status;
600033     void *pstatus;
600034     int i;
600035     //
600036     //
600037     //
600038     uid = geteuid ();
600039     //
600040     // Load processes, reading the keyboard.
600041     //
600042     while (1)
600043     {
600044         if (uid == 0)
600045         {
600046             printf ("# ");
600047         }
600048         else
600049         {
600050             printf ("$ ");
600051         }
600052         //
600053         pstatus = fgets (buffer_cmd, (ARG_MAX / 2), stdin);
600054         if (pstatus == NULL)
600055         {
600056             if (errno)
600057             {
600058                 perror (NULL);
600059                 continue;
600060             }
600061             else
600062             {
600063                 //
600064                 // End of file, like ^D.
600065                 //
600066                 return (0);
600067             }

```

```

600068     }
600069     //
600070     i = strlen (buffer_cmd);
600071     if (i > 0 && buffer_cmd[i - 1] == '\n')
600072     {
600073         buffer_cmd[i - 1] = '\0';
600074     }
600075     //
600076     // Clear 'argv_cmd[]';
600077     //
600078     for (argc_cmd = 0; argc_cmd < (ARG_MAX / 16);
600079         argc_cmd++)
600080     {
600081         argv_cmd[argc_cmd] = NULL;
600082     }
600083     //
600084     // Initialize the command scan.
600085     //
600086     argv_cmd[0] = strtok (buffer_cmd, " \t");
600087     //
600088     // Verify: if the input is not valid, loop
600089     // again.
600090     //
600091     if (argv_cmd[0] == NULL)
600092     {
600093         continue;
600094     }
600095     //
600096     // Find the arguments.
600097     //
600098     for (argc_cmd = 1;
600099         argc_cmd < ((ARG_MAX / 16) - 1)
600100         && argv_cmd[argc_cmd - 1] != NULL; argc_cmd++)
600101     {
600102         argv_cmd[argc_cmd] = strtok (NULL, " \t");
600103     }
600104     //
600105     // If there are too many arguments, show a
600106     // message and continue.
600107     //
600108     if (argv_cmd[argc_cmd - 1] != NULL)
600109     {
600110         errset (E2BIG); // Argument list too
600111         // long.
600112         perror (NULL);
600113         continue;
600114     }
600115     //
600116     // Correct the value for 'argc_cmd', because
600117     // actually
600118     // it counts also the NULL element.
600119     //
600120     argc_cmd--;
600121     //
600122     // Verify if it is an internal command.
600123     //
600124     if (strcmp (argv_cmd[0], "exit") == 0)
600125     {
600126         return (0);
600127     }
600128     else if (strcmp (argv_cmd[0], "cd") == 0)
600129     {
600130         sh_cd (argc_cmd, argv_cmd);
600131         continue;
600132     }
600133     else if (strcmp (argv_cmd[0], "pwd") == 0)
600134     {
600135         sh_pwd (argc_cmd, argv_cmd);
600136         continue;
600137     }
600138     else if (strcmp (argv_cmd[0], "umask") == 0)
600139     {
600140         sh_umask (argc_cmd, argv_cmd);
600141         continue;
600142     }
600143     //
600144     // It should be a program to run.
600145     //
600146     pid_cmd = fork ();
600147     if (pid_cmd == -1)
600148     {
600149         printf ("%s: cannot run command", argv[0]);
600150         perror (NULL);
600151     }
600152     else if (pid_cmd == 0)
600153     {
600154         execvp (argv_cmd[0], argv_cmd);

```

```

600155         perror (NULL);
600156         exit (0);
600157     }
600158     while (1)
600159     {
600160         pid_dead = wait (&status);
600161         if (pid_dead == pid_cmd)
600162         {
600163             break;
600164         }
600165     }
600166     printf ("pid %i terminated with status %i.\n",
600167           (int) pid_dead, status);
600168     }
600169 }
600170
600171 //-----
600172 static void
600173 sh_cd (int argc, char *argv[])
600174 {
600175     int status;
600176     //
600177     if (argc != 2)
600178     {
600179         errset (EINVAL); // Invalid argument.
600180         perror (NULL);
600181         return;
600182     }
600183     //
600184     status = chdir (argv[1]);
600185     if (status != 0)
600186     {
600187         perror (NULL);
600188     }
600189     return;
600190 }
600191
600192 //-----
600193 static void
600194 sh_pwd (int argc, char *argv[])
600195 {
600196     char path[PATH_MAX];
600197     void *pstatus;
600198     //
600199     if (argc != 1)
600200     {
600201         errset (EINVAL); // Invalid argument.
600202         perror (NULL);
600203         return;
600204     }
600205     //
600206     // Get the current directory.
600207     //
600208     pstatus = getcwd (path, (size_t) PATH_MAX);
600209     if (pstatus == NULL)
600210     {
600211         perror (NULL);
600212     }
600213     else
600214     {
600215         printf ("%s\n", path);
600216     }
600217     return;
600218 }
600219
600220 //-----
600221 static void
600222 sh_umask (int argc, char *argv[])
600223 {
600224     sysmsg_uarea_t msg;
600225     char *m; // Index inside the umask octal
600226     // string.
600227     int mask;
600228     int digit;
600229     //
600230     if (argc > 2)
600231     {
600232         errset (EINVAL); // Invalid argument.
600233         perror (NULL);
600234         return;
600235     }
600236     //
600237     // If no argument is available, the umask is shown,
600238     // with a direct
600239     // system call.
600240     //
600241     if (argc == 1)

```

```

6000242 {
6000243     sys (SYS_UAREA, &msg, (sizeof msg));
6000244     printf ("%04o\n", msg.umask);
6000245     return;
6000246 }
6000247 //
6000248 // Get the mask: must be the first argument.
6000249 //
6000250 for (mask = 0, m = argv[1]; *m != 0; m++)
6000251 {
6000252     digit = (*m - '0');
6000253     if (digit < 0 || digit > 7)
6000254     {
6000255         errset (EINVAL); // Invalid argument.
6000256         perror (NULL);
6000257         return;
6000258     }
6000259     mask = mask * 8 + digit;
6000260 }
6000261 //
6000262 // Set the umask and return.
6000263 //
6000264 umask (mask);
6000265 return;
6000266 }

```

### 96.1.36 applic/t\_fcntl.c

« Si veda la sezione 86.25.

```

6010001 #include <stdio.h>
6010002 #include <unistd.h>
6010003 #include <stdlib.h>
6010004 #include <sys/wait.h>
6010005 #include <fcntl.h>
6010006 //-----
6010007 int
6010008 main (void)
6010009 {
6010010     int status;
6010011
6010012
6010013     printf ("opzione O_NONBLOCK=%08x\n", O_NONBLOCK);
6010014
6010015     fcntl (STDIN_FILENO, F_SETFL, O_NONBLOCK);
6010016
6010017     status = fcntl (STDIN_FILENO, F_GETFL);
6010018
6010019
6010020
6010021     printf ("fcntl (STDIN_FILENO, F_GETFL) == %08x\n",
6010022           status);
6010023
6010024     return (0);
6010025 }
6010026 }

```

### 96.1.37 applic/t\_fifo.c

« Si veda la sezione 86.25.

```

6020001 #include <stdio.h>
6020002 #include <unistd.h>
6020003 #include <stdlib.h>
6020004 #include <sys/wait.h>
6020005 #include <signal.h>
6020006 #include <sys/wait.h>
6020007 #include <string.h>
6020008 #include <fcntl.h>
6020009 #include <sys/stat.h>
6020010 //-----
6020011 int
6020012 main (void)
6020013 {
6020014     int fd;
6020015     pid_t child;
6020016     char buffer;
6020017     char *message =
6020018         "ciao a tutti voi amici vicini e lontani\n";
6020019     int i;
6020020     size_t size;
6020021     ssize_t written;
6020022     int status;
6020023     //
6020024     //
6020025     //
6020026     unlink ("/tmp/fifo");

```

```

6020027 //
6020028     status =
6020029         mknod ("/tmp/fifo", S_IFIFO | S_IRUSR | S_IWUSR, 0);
6020030     if (status != 0)
6020031     {
6020032         perror ("mknod fifo");
6020033         exit (EXIT_FAILURE);
6020034     }
6020035     //
6020036     //
6020037     //
6020038     child = fork ();
6020039     if (child == -1)
6020040     {
6020041         perror ("fork");
6020042         exit (EXIT_FAILURE);
6020043     }
6020044     //
6020045     //
6020046     //
6020047     if (child == 0)
6020048     {
6020049         //
6020050         // This is the child and it have to read the
6020051         // fifo.
6020052         //
6020053         fd = open ("/tmp/fifo", O_RDONLY);
6020054         if (fd < 0)
6020055         {
6020056             perror ("fifo read open");
6020057             exit (EXIT_FAILURE);
6020058         }
6020059         //
6020060         // Read one byte at the time, as long as there
6020061         // is
6020062         // something to read.
6020063         //
6020064         while (read (fd, &buffer, 1) > 0)
6020065         {
6020066             write (STDOUT_FILENO, &buffer, 1);
6020067         }
6020068         //
6020069         // Close the fifo and exit the child.
6020070         //
6020071         close (fd);
6020072         //
6020073         exit (EXIT_SUCCESS);
6020074     }
6020075     else
6020076     {
6020077         //
6020078         // This is the parent process and it writes to
6020079         // the FIFO.
6020080         //
6020081         fd = open ("/tmp/fifo", O_WRONLY);
6020082         if (fd < 0)
6020083         {
6020084             perror ("fifo write open");
6020085             exit (EXIT_FAILURE);
6020086         }
6020087         //
6020088         while (1)
6020089         {
6020090             for (i = 0, written = 0, size =
6020091                  strlen (message); i < strlen (message);
6020092                  i += written, size -= written)
6020093             {
6020094                 written = write (fd, &message[i], size);
6020095                 if (written < 0)
6020096                 {
6020097                     perror ("pipe");
6020098                     close (fd);
6020099                     wait (NULL); // Wait for child.
6020100                     exit (EXIT_FAILURE);
6020101                 }
6020102             }
6020103         }
6020104         close (fd); /* Reader will see EOF */
6020105         wait (NULL); /* Wait for child */
6020106         exit (EXIT_SUCCESS);
6020107     }
6020108     //
6020109     return (0);
6020110 }

```

## 96.1.38 applic/t\_grp.c

Si veda la sezione 86.25.

```

6030001 #include <stdio.h>
6030002 #include <grp.h>
6030003 #include <pwd.h>
6030004 #include <unistd.h>
6030005 #include <stdlib.h>
6030006 #include <sys/wait.h>
6030007 #include <signal.h>
6030008 #include <sys/wait.h>
6030009 #include <string.h>
6030010 #include <fcntl.h>
6030011 #include <sys/stat.h>
6030012 //-----
6030013 int
6030014 main (void)
6030015 {
6030016     struct passwd *pw;
6030017     struct group *gr;
6030018     int i;
6030019
6030020     pw = getpwuid ((uid_t) 1001);
6030021
6030022     if (pw == NULL)
6030023     {
6030024         perror (NULL);
6030025         exit (0);
6030026     }
6030027
6030028     printf ("%s:%s:%i:\n", pw->pw_name, pw->pw_passwd,
6030029            pw->pw_uid, pw->pw_gid);
6030030
6030031     gr = getgrgid ((gid_t) 233);
6030032
6030033     if (gr == NULL)
6030034     {
6030035         perror (NULL);
6030036         exit (0);
6030037     }
6030038
6030039     printf ("%s:%s:%i:", gr->gr_name, gr->gr_passwd,
6030040            gr->gr_gid);
6030041     for (i = 0; i < 32 && gr->gr_mem[i] != NULL; i++)
6030042     {
6030043         printf ("%s,", gr->gr_mem[i]);
6030044     }
6030045
6030046     return (0);
6030047 }

```

## 96.1.39 applic/t\_nc.c

Si veda la sezione 86.25.

```

6040001 #include <sys/stat.h>
6040002 #include <sys/types.h>
6040003 #include <unistd.h>
6040004 #include <stdlib.h>
6040005 #include <fcntl.h>
6040006 #include <errno.h>
6040007 #include <signal.h>
6040008 #include <stdio.h>
6040009 #include <string.h>
6040010 #include <limits.h>
6040011 #include <libgen.h>
6040012 #include <arpa/inet.h>
6040013 #include <sys/socket.h>
6040014 #include <stdint.h>
6040015 #include <stdbool.h>
6040016 #include <fcntl.h>
6040017 //-----
6040018 static void usage (void);
6040019 char buffer[BUFSIZ];
6040020 //-----
6040021 int
6040022 main (int argc, char *argv[], char *envp[])
6040023 {
6040024     bool option_l = 0;
6040025     bool option_u = 0;
6040026     int opt;
6040027     //extern char *optarg;           // not used.
6040028     extern int optind;
6040029     extern int optopt;
6040030     //
6040031     int status;
6040032     int sfdn;

```

```

6040033     int sfdn2;
6040034     struct sockaddr_in sa_local;
6040035     struct sockaddr_in sa_remote;
6040036     socklen_t sa_remote_size = sizeof (struct sockaddr_in);
6040037     ssize_t read_size;
6040038     ssize_t sent_size;
6040039     ssize_t recv_size;
6040040     char *addr = NULL;
6040041     char *port = NULL;
6040042     bool can_rx = 1;
6040043     bool can_tx = 1;
6040044     //
6040045     // Check for options.
6040046     //
6040047     while ((opt = getopt (argc, argv, "ul")) != -1)
6040048     {
6040049         switch (opt)
6040050         {
6040051             case 'l':
6040052                 option_l = 1;
6040053                 break;
6040054             case 'u':
6040055                 option_u = 1;
6040056                 break;
6040057             case '?':
6040058                 fprintf (stderr, "Unknown option -%c.\n", optopt);
6040059                 usage ();
6040060                 return (1);
6040061             break;
6040062             case ':':
6040063                 fprintf (stderr,
6040064                        "Missing argument for option -%c\n",
6040065                        optopt);
6040066                 usage ();
6040067                 return (1);
6040068             break;
6040069             default:
6040070                 fprintf (stderr,
6040071                        "Getopt problem: unknown option %c\n",
6040072                        opt);
6040073                 usage ();
6040074                 return (1);
6040075             }
6040076         }
6040077     //
6040078     // Arguments.
6040079     //
6040080     if (optind == (argc - 2))
6040081     {
6040082         //
6040083         // There are exactly two arguments: destination
6040084         // address and port.
6040085         //
6040086         addr = argv[argc - 2];
6040087         port = argv[argc - 1];
6040088     }
6040089     else
6040090     {
6040091         //
6040092         // Arguments wrong!
6040093         //
6040094         usage ();
6040095         return (2);
6040096     }
6040097     //
6040098     // Set the local or the remote address.
6040099     //
6040100     if (option_l)
6040101     {
6040102         //
6040103         // Address and port are local.
6040104         //
6040105         sa_local.sin_family = AF_INET;
6040106         sa_local.sin_port = htons (atoi (port));
6040107         inet_pton (AF_INET, addr, &sa_local.sin_addr.s_addr);
6040108     }
6040109     else
6040110     {
6040111         //
6040112         // Address and port are remote.
6040113         //
6040114         sa_remote.sin_family = AF_INET;
6040115         sa_remote.sin_port = htons (atoi (port));
6040116         inet_pton (AF_INET, addr, &sa_remote.sin_addr.s_addr);
6040117     }
6040118     //
6040119     // Open the socket.

```

```

6040120 //
6040121 if (option_u)
6040122 {
6040123     sfdn = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
6040124 }
6040125 else
6040126 {
6040127     sfdn = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
6040128 }
6040129 if (sfdn < 0)
6040130 {
6040131     perror (NULL);
6040132     return (3);
6040133 }
6040134 //
6040135 // Set it listening or connect.
6040136 //
6040137 if (option_l)
6040138 {
6040139     //
6040140     // Bind the local 'sa' location.
6040141     //
6040142     status =
6040143         bind (sfdn, (struct sockaddr *) &sa_local,
6040144             sizeof (sa_local));
6040145     if (status < 0)
6040146     {
6040147         perror (NULL);
6040148         close (sfdn);
6040149         return (4);
6040150     }
6040151     //
6040152     // Listen (TCP) or wait the first packet (UDP).
6040153     //
6040154     if (option_u)
6040155     {
6040156         //
6040157         // Instead of listening, we use the function
6040158         // 'recvfrom()',
6040159         // to get the remote address and port.
6040160         //
6040161         recv_size =
6040162             recvfrom (sfdn, &buffer,
6040163                     (size_t) BUFSIZ - 1, 0,
6040164                     (struct sockaddr *) &sa_remote,
6040165                     &sa_remote_size);
6040166         if (recv_size < 0)
6040167         {
6040168             perror (NULL);
6040169             close (sfdn);
6040170             return (4);
6040171         }
6040172         //
6040173         // Now connect the remote destination
6040174         //
6040175         status =
6040176             connect (sfdn,
6040177                    (struct sockaddr *) &sa_remote,
6040178                    sizeof (sa_remote));
6040179         if (status < 0)
6040180         {
6040181             perror (NULL);
6040182             close (sfdn);
6040183             return (7);
6040184         }
6040185         //
6040186         // And show what was received as a first
6040187         // packet.
6040188         //
6040189         buffer[recv_size] = 0;
6040190         printf ("%s", buffer);
6040191     }
6040192     else
6040193     {
6040194         //
6040195         // TCP: listen.
6040196         //
6040197         status = listen (sfdn, 1);
6040198         if (status < 0)
6040199         {
6040200             perror (NULL);
6040201             close (sfdn);
6040202             return (5);
6040203         }
6040204         //
6040205         // Accept.
6040206         //

```

```

6040207     sfdn2 =
6040208         accept (sfdn,
6040209                (struct sockaddr *) &sa_remote,
6040210                &sa_remote_size);
6040211     if (sfdn2 < 0)
6040212     {
6040213         perror (NULL);
6040214         close (sfdn);
6040215         return (6);
6040216     }
6040217     //
6040218     // Close listening socket.
6040219     //
6040220     close (sfdn);
6040221     //
6040222     // Variable 'sfdn' will be the new socket.
6040223     //
6040224     sfdn = sfdn2;
6040225 }
6040226 }
6040227 else
6040228 {
6040229     //
6040230     // Connect the remote destination.
6040231     //
6040232     status =
6040233         connect (sfdn, (struct sockaddr *) &sa_remote,
6040234                 sizeof (sa_remote));
6040235     if (status < 0)
6040236     {
6040237         perror (NULL);
6040238         close (sfdn);
6040239         return (7);
6040240     }
6040241 }
6040242 //
6040243 // Define the standard input non blocking.
6040244 //
6040245 status = fcntl (STDIN_FILENO, F_SETFL, O_NONBLOCK);
6040246 if (status < 0)
6040247 {
6040248     perror (NULL);
6040249     return (8);
6040250 }
6040251 //
6040252 // Will read from the remote and show to the screen.
6040253 //
6040254 while (can_rx || can_tx)
6040255 {
6040256     if (can_rx)
6040257     {
6040258         recv_size =
6040259             recv (sfdn, &buffer, (size_t) BUFSIZ - 1, 0);
6040260         //
6040261         // recv_size = read (sfdn, &buffer, (size_t) BUFSIZ-1);
6040262         if (recv_size < 0)
6040263         {
6040264             if (errno == EAGAIN || errno == EWOULDBLOCK)
6040265             {
6040266                 ;
6040267             }
6040268             else
6040269             {
6040270                 perror (NULL);
6040271                 close (sfdn);
6040272                 return (10);
6040273             }
6040274         }
6040275         else if (recv_size == 0)
6040276         {
6040277             //
6040278             // End of stream.
6040279             //
6040280             can_rx = 0;
6040281             printf ("--end of receive stream--\n");
6040282         }
6040283         else
6040284         {
6040285             buffer[recv_size] = 0;
6040286             printf ("%s", buffer);
6040287         }
6040288     }
6040289     if (can_tx)
6040290     {
6040291         read_size = read (STDIN_FILENO, buffer, BUFSIZ);
6040292         if (read_size < 0)
6040293         {
6040294             if (errno == EAGAIN || errno == EWOULDBLOCK)

```

```

6040294     {
6040295     ;
6040296     }
6040297     else
6040298     {
6040299         perror (NULL);
6040300         close (sfdn);
6040301         return (11);
6040302     }
6040303     }
6040304     else if (read_size == 0)
6040305     {
6040306         //
6040307         // End of input.
6040308         //
6040309         printf ("--closing send stream--\n");
6040310         can_tx = 0;
6040311     }
6040312     else
6040313     {
6040314         //
6040315         // Send it.
6040316         //
6040317         sent_size =
6040318             send (sfdn, &buffer, (size_t) read_size, 0);
6040319         if (sent_size < 0)
6040320         {
6040321             if (errno == EAGAIN
6040322                 || errno == EWOULDBLOCK)
6040323             {
6040324                 ;
6040325             }
6040326             else
6040327             {
6040328                 perror (NULL);
6040329                 close (sfdn);
6040330                 return (12);
6040331             }
6040332         }
6040333     }
6040334     }
6040335     }
6040336     //
6040337     // All done.
6040338     //
6040339     close (sfdn);
6040340     return (0);
6040341 }
6040342
6040343 //-----
6040344 static void
6040345 usage (void)
6040346 {
6040347     fprintf (stderr,
6040348             "os32 netcat usage:\n"
6040349             "\n"
6040350             "nc [-u][-l] ADDRESS PORT\n"
6040351             "\n"
6040352             "-u      Use UDP protocol instead of TCP.\n"
6040353             "-l      Listen for incoming connection requests.\n"
6040354             "ADDRESS IPv4 numeric address; if option -l is used, this
6040355             is the local address, otherwise it is the remote
6040356             address.\n"
6040357             "PORT   TCP or UDP port; if option -l is used, this is\
6040358             local address, otherwise it is the remote\n"
6040359             "       address.\n");
6040360 }

```

### 96.1.40 applic/t\_ping2.c

« Si veda la sezione 86.25.

```

6050001 #include <stdio.h>
6050002 #include <sys/types.h>
6050003 // #include <arpa/inet.h>
6050004 #include <sys/socket.h>
6050005 #include <unistd.h>
6050006 #include <errno.h>
6050007 //-----
6050008 int
6050009 main (void)
6050010 {
6050011     int i;
6050012     int sfdn;
6050013     struct sockaddr_in sa;
6050014     ssize_t spediti;
6050015     ssize_t ricevuti;

```

```

6050016     int status;
6050017     uint8_t buffer[100];
6050018     uint8_t packet[] =
6050019     { 0x45, 0x00, 0x00, 0x22, 0x00, 0x00, 0x00, 0x40, 0x00,
6050020       0x40, 0x01, 0x3c, 0xd9, 0x7f, 0x00, 0x00, 0x01,
6050021       0x7f, 0x00, 0x00, 0x01, 'c', 'i', 'a', 'o', ' ',
6050022       'a', 'm', 'o', 'r', 'e', ' ', 'm', 'i', 'o'
6050023     };
6050024
6050025     sa.sin_family = AF_INET;
6050026     sa.sin_port = 0;
6050027     // sa.sin_addr.s_addr=htonl (0xAC15FEFE); //172.21.254.254
6050028     sa.sin_addr.s_addr = htonl (0xAC150B12); // 172.21.11.18
6050029
6050030     errno = 0;
6050031     sfdn = socket (AF_INET, SOCK_RAW, IPPROTO_ICMP);
6050032     perror (NULL);
6050033
6050034     errno = 0;
6050035     status =
6050036         connect (sfdn, (struct sockaddr *) &sa, sizeof (sa));
6050037     perror (NULL);
6050038
6050039     errno = 0;
6050040     spediti = send (sfdn, packet, 34, 0);
6050041     printf ("scritti %i\n", spediti);
6050042     perror (NULL);
6050043
6050044     ricevuti = 10;
6050045     while (ricevuti > 0)
6050046     {
6050047         errno = 0;
6050048         ricevuti = recv (sfdn, buffer, (size_t) 30, 0);
6050049         printf ("ricevuti=%i\n", (int) ricevuti);
6050050         perror (NULL);
6050051
6050052         if (ricevuti > 0)
6050053         {
6050054             for (i = 0; i < ricevuti; i++)
6050055             {
6050056                 printf ("%02x", (unsigned int) buffer[i]);
6050057             }
6050058         }
6050059     }
6050060
6050061     close (sfdn);
6050062     return (0);
6050063 }

```

### 96.1.41 applic/t\_pipe.c

« Si veda la sezione 86.25.

```

6060001 #include <stdio.h>
6060002 #include <unistd.h>
6060003 #include <stdlib.h>
6060004 #include <sys/wait.h>
6060005 #include <signal.h>
6060006 #include <sys/wait.h>
6060007 #include <stdio.h>
6060008 #include <stdlib.h>
6060009 #include <string.h>
6060010 //-----
6060011 int
6060012 main (void)
6060013 {
6060014     int pipefd[2];
6060015     pid_t child;
6060016     char buffer;
6060017     char *message =
6060018         "ciao a tutti voi amici vicini e lontani\n";
6060019     int i;
6060020     size_t size;
6060021     ssize_t written;
6060022     //
6060023     //
6060024     //
6060025     if (pipe (pipefd) == -1)
6060026     {
6060027         perror ("pipe");
6060028         exit (EXIT_FAILURE);
6060029     }
6060030     //
6060031     //
6060032     //
6060033     child = fork ();
6060034     if (child == -1)

```

```

600035 {
600036     perror ("fork");
600037     exit (EXIT_FAILURE);
600038 }
600039 //
600040 //
600041 //
600042 if (child == 0)
600043 {
600044     //
600045     // This is the child and it have to read the
600046     // pipe:
600047     // Close the write end of the pipe.
600048     //
600049     close (pipefd[1]);
600050     //
600051     // Read one byte at the time, as long as there
600052     // is
600053     // something to read.
600054     //
600055     while (read (pipefd[0], &buffer, 1) > 0)
600056     {
600057         write (STDOUT_FILENO, &buffer, 1);
600058     }
600059     //
600060     // Close the pipe and exit the child.
600061     //
600062     close (pipefd[0]);
600063     //
600064     exit (EXIT_SUCCESS);
600065 }
600066 else
600067 {
600068     //
600069     // This is the parent process, and the read end
600070     // of
600071     // pipe is closed.
600072     //
600073     close (pipefd[0]);
600074     //
600075     while (1)
600076     {
600077         for (i = 0, written = 0, size =
600078             strlen (message); i < strlen (message);
600079              i += written, size -= written)
600080         {
600081             written =
600082                 write (pipefd[1], &message[i], size);
600083             if (written < 0)
600084             {
600085                 perror ("pipe");
600086                 close (pipefd[1]);
600087                 wait (NULL); // Wait for child.
600088                 exit (EXIT_FAILURE);
600089             }
600090         }
600091     }
600092     close (pipefd[1]); // Reader will see EOF */
600093     wait (NULL); // Wait for child */
600094     exit (EXIT_SUCCESS);
600095 }
600096 //
600097 return (0);
600098 }

```

### 96.1.42 applic/t\_read.c

« Si veda la sezione 86.25.

```

607001 #include <sys/stat.h>
607002 #include <sys/types.h>
607003 #include <unistd.h>
607004 #include <stdlib.h>
607005 #include <fcntl.h>
607006 #include <errno.h>
607007 #include <signal.h>
607008 #include <stdio.h>
607009 #include <string.h>
607010 #include <limits.h>
607011 #include <libgen.h>
607012 #include <arpa/inet.h>
607013 #include <sys/socket.h>
607014 #include <stdint.h>
607015 #include <stdbool.h>
607016 #include <fcntl.h>
607017
607018 char buffer[BUFSIZ];

```

```

607019
607020 //-----
607021 int
607022 main (int argc, char *argv[], char *envp[])
607023 {
607024     int status;
607025     ssize_t read_size;
607026
607027
607028     //
607029     // Define the standard input non blocking.
607030     //
607031     status = fcntl (STDIN_FILENO, F_SETFL, O_NONBLOCK);
607032     if (status < 0)
607033     {
607034         perror (NULL);
607035         return (2);
607036     }
607037
607038
607039     read_size = read (STDIN_FILENO, buffer, BUFSIZ);
607040     if (read_size < 0)
607041     {
607042         if (errno == EAGAIN || errno == EWOULDBLOCK)
607043         {
607044             printf ("nulla da leggere per ora\n");
607045         }
607046         else
607047         {
607048             perror (NULL);
607049             return (0);
607050         }
607051     }
607052     else
607053     {
607054         buffer[read_size] = 0;
607055         printf ("letto: %s\n", buffer);
607056     }
607057     printf ("finito\n");
607058     return (0);
607059 }

```

### 96.1.43 applic/t\_ret.c

« Si veda la sezione 86.25.

```

608001 #include <stdlib.h>
608002 //-----
608003 int
608004 main (void)
608005 {
608006     // exit (1);
608007     return (1);
608008 }

```

### 96.1.44 applic/t\_rx\_udp.c

« Si veda la sezione 86.25.

```

609001 #include <sys/stat.h>
609002 #include <sys/types.h>
609003 #include <unistd.h>
609004 #include <stdlib.h>
609005 #include <fcntl.h>
609006 #include <errno.h>
609007 #include <signal.h>
609008 #include <stdio.h>
609009 #include <string.h>
609010 #include <limits.h>
609011 #include <libgen.h>
609012 #include <arpa/inet.h>
609013 #include <sys/socket.h>
609014 #include <stdint.h>
609015 #include <stdbool.h>
609016 //-----
609017 static void usage (void);
609018 //-----
609019 int
609020 main (int argc, char *argv[], char *envp[])
609021 {
609022     int status;
609023     int sfdn;
609024     struct sockaddr_in sa_local;
609025     ssize_t recv_size;
609026     char buffer[BUFSIZ];
609027     char *addr = NULL;
609028     char *port = NULL;

```

```

609029 //
609030 // Arguments.
609031 //
609032 if (argc == 3)
609033 {
609034 //
609035 // There are exactly two arguments: destination
609036 // address and port.
609037 //
609038 addr = argv[1];
609039 port = argv[2];
609040 }
609041 else
609042 {
609043 //
609044 // Arguments wrong!
609045 //
609046 usage ();
609047 return (4);
609048 }
609049 //
609050 // Define the destination 'sa_local'
609051 //
609052 sa_local.sin_family = AF_INET;
609053 sa_local.sin_port = htons (atoi (port));
609054 inet_pton (AF_INET, addr, &sa_local.sin_addr.s_addr);
609055 //
609056 // Open the socket.
609057 //
609058 sfdn = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
609059 if (sfdn < 0)
609060 {
609061 perror (NULL);
609062 return (5);
609063 }
609064 //
609065 // Bind the local 'sa' location.
609066 //
609067 status = bind (sfdn, (struct sockaddr *) &sa_local,
609068 sizeof (sa_local));
609069 if (status < 0)
609070 {
609071 perror (NULL);
609072 close (sfdn);
609073 return (7);
609074 }
609075 //
609076 // Will read from the remote and show to the screen.
609077 //
609078 while (1)
609079 {
609080 recv_size = read (sfdn, &buffer, (size_t) BUFSIZ - 1);
609081 if (recv_size < 0)
609082 {
609083 perror (NULL);
609084 close (sfdn);
609085 return (10);
609086 }
609087 buffer[recv_size] = 0;
609088 printf ("%s", buffer);
609089 }
609090 //
609091 // All done.
609092 //
609093 return (0);
609094 }
609095 //-----
609096 static void
609097 usage (void)
609098 {
609099 fprintf (stderr, "Usage: rx_udp LOCAL_ADDR LOCAL_PORT\n");
609100 }

```

## 96.1.45 applic/t\_scr.c

« Si veda la sezione 86.25.

```

610001 #include <unistd.h>
610002 #include <stdio.h>
610003 #include <fcntl.h>
610004 #include <unistd.h>
610005 #include <stdlib.h>
610006 //-----
610007 int
610008 main (int argc, char *argv[], char *envp[])
610009 {

```

```

610010 FILE *screen;
610011 int status;
610012
610013 screen = fopen ("/dev/tty", "w");
610014 if (screen == NULL)
610015 {
610016 printf ("[%s] Cannot open \"/dev/tty\" ", argv[0]);
610017 perror (NULL);
610018 exit (0);
610019 }
610020
610021 status = fseek (screen, (long) 1000, SEEK_SET);
610022
610023 fprintf (screen, "ciao status: %i ciao", status);
610024 perror (NULL);
610025
610026 fclose (screen);
610027 return (0);
610028 }

```

## 96.1.46 applic/t\_setjmp.c

Si veda la sezione 86.25. »

```

610001 #include <stdio.h>
610002 #include <setjmp.h>
610003 //-----
610004
610005 jmp_buf env;
610006
610007 void
610008 prova3 (void)
610009 {
610010 printf ("funzione prova3\n");
610011 longjmp (env, 1);
610012 printf ("funzione prova3 post\n");
610013 }
610014
610015 void
610016 prova2 (void)
610017 {
610018 printf ("funzione prova2\n");
610019 prova3 ();
610020 printf ("funzione prova2 post\n");
610021 }
610022
610023 void
610024 proval (void)
610025 {
610026 printf ("funzione proval\n");
610027 prova2 ();
610028 printf ("funzione proval post\n");
610029 }
610030
610031 int
610032 main (int argc, char *argv[], char *envp[])
610033 {
610034 int val;
610035 //
610036 printf ("prima\n");
610037 //
610038 val = setjmp (env);
610039 //
610040 printf ("dopo setjmp val=%i\n", val);
610041 //
610042 if (val != 0)
610043 return (0);
610044
610045 proval ();
610046
610047 return (0);
610048 }

```

## 96.1.47 applic/t\_sig.c

Si veda la sezione 86.25. »

```

612001 #include <stdio.h>
612002 #include <unistd.h>
612003 #include <stdlib.h>
612004 #include <sys/wait.h>
612005 #include <signal.h>
612006 //-----
612007 void
612008 signal_handler (int signal)
612009 {
612010 printf ("Hello! I have caught the signal %i.\n", signal);

```



```

6120011 }
6120012
6120013 //-----
6120014 int
6120015 main (void)
6120016 {
6120017     signal (SIGHUP, signal_handler);
6120018     signal (SIGINT, signal_handler);
6120019     signal (SIGQUIT, signal_handler);
6120020     signal (SIGILL, signal_handler);
6120021     signal (SIGABRT, signal_handler);
6120022     signal (SIGFPE, signal_handler);
6120023     signal (SIGKILL, signal_handler);
6120024     signal (SIGSEGV, signal_handler);
6120025     signal (SIGPIPE, signal_handler);
6120026     signal (SIGALRM, signal_handler);
6120027     signal (SIGTERM, signal_handler);
6120028     signal (SIGSTOP, signal_handler);
6120029     signal (SIGTSTP, signal_handler);
6120030     signal (SIGCONT, signal_handler);
6120031     signal (SIGCHLD, signal_handler);
6120032     signal (SIGTTIN, signal_handler);
6120033     signal (SIGTTOU, signal_handler);
6120034     signal (SIGUSR1, signal_handler);
6120035     signal (SIGUSR2, signal_handler);
6120036     //
6120037     while (1)
6120038     {
6120039         sleep (1);
6120040         printf ("ciao!\n");
6120041     }
6120042     //
6120043     return (0);
6120044 }

```

## 96.1.48 applic/t\_sig2.c

Si veda la sezione 86.25.

```

6130001 #include <stdio.h>
6130002 #include <unistd.h>
6130003 #include <stdlib.h>
6130004 #include <sys/wait.h>
6130005 #include <signal.h>
6130006 //-----
6130007 void
6130008 signal_handler (int signal)
6130009 {
6130010     printf ("Hello! I have caught the signal %i.\n", signal);
6130011 }
6130012 //-----
6130013 int
6130014 main (void)
6130015 {
6130016     //
6130017     while (1)
6130018     {
6130019         signal (SIGHUP, signal_handler);
6130020         signal (SIGINT, signal_handler);
6130021         signal (SIGQUIT, signal_handler);
6130022         signal (SIGILL, signal_handler);
6130023         signal (SIGABRT, signal_handler);
6130024         signal (SIGFPE, signal_handler);
6130025         signal (SIGKILL, signal_handler);
6130026         signal (SIGSEGV, signal_handler);
6130027         signal (SIGPIPE, signal_handler);
6130028         signal (SIGALRM, signal_handler);
6130029         signal (SIGTERM, signal_handler);
6130030         signal (SIGSTOP, signal_handler);
6130031         signal (SIGTSTP, signal_handler);
6130032         signal (SIGCONT, signal_handler);
6130033         signal (SIGCHLD, signal_handler);
6130034         signal (SIGTTIN, signal_handler);
6130035         signal (SIGTTOU, signal_handler);
6130036         signal (SIGUSR1, signal_handler);
6130037         signal (SIGUSR2, signal_handler);
6130038         printf ("ciao!\n");
6130039         sleep (1);
6130040     }
6130041     //
6130042     return (0);
6130043 }

```

## 96.1.49 applic/t\_tx\_tcp.c

Si veda la sezione 86.25.

```

6140001 #include <sys/stat.h>
6140002 #include <sys/types.h>
6140003 #include <unistd.h>
6140004 #include <stdlib.h>
6140005 #include <fcntl.h>
6140006 #include <errno.h>
6140007 #include <signal.h>
6140008 #include <stdio.h>
6140009 #include <string.h>
6140010 #include <limits.h>
6140011 #include <libgen.h>
6140012 #include <arpa/inet.h>
6140013 #include <sys/socket.h>
6140014 #include <stdint.h>
6140015 #include <stdbool.h>
6140016 //-----
6140017 static void usage (void);
6140018 //-----
6140019 int
6140020 main (int argc, char *argv[], char *envp[])
6140021 {
6140022     int status;
6140023     int sfdn;
6140024     struct sockaddr_in sa;
6140025     ssize_t read_size;
6140026     ssize_t sent_size;
6140027     char buffer[BUFSIZ];
6140028     char *addr = NULL;
6140029     char *port = NULL;
6140030     //
6140031     // Arguments.
6140032     //
6140033     if (argc == 3)
6140034     {
6140035         //
6140036         // There are exactly two arguments: destination
6140037         // address and port.
6140038         //
6140039         addr = argv[1];
6140040         port = argv[2];
6140041     }
6140042     else
6140043     {
6140044         //
6140045         // Arguments wrong!
6140046         //
6140047         usage ();
6140048         return (4);
6140049     }
6140050     //
6140051     // Define the destination 'sa'
6140052     //
6140053     sa.sin_family = AF_INET;
6140054     sa.sin_port = htons (atoi (port));
6140055     inet_pton (AF_INET, addr, &sa.sin_addr.s_addr);
6140056     //
6140057     //
6140058     // Open the socket.
6140059     //
6140060     sfdn = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
6140061     if (sfdn < 0)
6140062     {
6140063         perror (NULL);
6140064         return (5);
6140065     }
6140066     //
6140067     // Connect the 'sa' destination
6140068     //
6140069     status =
6140070     connect (sfdn, (struct sockaddr *) &sa, sizeof (sa));
6140071     if (status < 0)
6140072     {
6140073         perror (NULL);
6140074         close (sfdn);
6140075         return (7);
6140076     }
6140077     //
6140078     // Will read from the standard input and send to the
6140079     // other
6140080     // side.
6140081     //
6140082     while (1)
6140083     {
6140084         read_size = read (STDIN_FILENO, buffer, BUFSIZ);
6140085         if (read_size < 0)

```

```

6140086     {
6140087         perror (NULL);
6140088         close (sfdn);
6140089         return (8);
6140090     }
6140091     if (read_size == 0)
6140092     {
6140093         close (sfdn);
6140094         return (0);
6140095     }
6140096     //
6140097     // Verify the 'stop' command.
6140098     //
6140099     if (strncmp (buffer, "stop\n", read_size) == 0)
6140100     {
6140101         printf ("closing send...\n");
6140102         close (sfdn);
6140103         return (0);
6140104     }
6140105     //
6140106     sent_size =
6140107     send (sfdn, &buffer, (size_t) read_size, 0);
6140108     if (sent_size < 0)
6140109     {
6140110         perror (NULL);
6140111         close (sfdn);
6140112         return (9);
6140113     }
6140114     printf ("sent %i bytes\n", (int) sent_size);
6140115 }
6140116 //
6140117 // All done.
6140118 //
6140119 return (0);
6140120 }
6140121
6140122 -----
6140123 static void
6140124 usage (void)
6140125 {
6140126     fprintf (stderr, "Usage: tx_tcp DEST_ADDR DEST_PORT\n");
6140127 }

```

### 96.1.50 applic/t\_tx\_udp.c

« Si veda la sezione 86.25.

```

6150001 #include <sys/stat.h>
6150002 #include <sys/types.h>
6150003 #include <unistd.h>
6150004 #include <stdlib.h>
6150005 #include <fcntl.h>
6150006 #include <errno.h>
6150007 #include <signal.h>
6150008 #include <stdio.h>
6150009 #include <string.h>
6150010 #include <limits.h>
6150011 #include <libgen.h>
6150012 #include <arpa/inet.h>
6150013 #include <sys/socket.h>
6150014 #include <stdint.h>
6150015 #include <stdbool.h>
6150016 -----
6150017 static void usage (void);
6150018 -----
6150019 int
6150020 main (int argc, char *argv[], char *envp[])
6150021 {
6150022     int status;
6150023     int sfdn;
6150024     struct sockaddr_in sa;
6150025     ssize_t read_size;
6150026     ssize_t sent_size;
6150027     char buffer[BUFSIZ];
6150028     char *addr = NULL;
6150029     char *port = NULL;
6150030     //
6150031     // Arguments.
6150032     //
6150033     if (argc == 3)
6150034     {
6150035         //
6150036         // There are exactly two arguments: destination
6150037         // address and port.
6150038         //
6150039         addr = argv[1];
6150040         port = argv[2];

```

```

6150041     }
6150042     else
6150043     {
6150044         //
6150045         // Arguments wrong!
6150046         //
6150047         usage ();
6150048         return (4);
6150049     }
6150050     //
6150051     // Define the destination 'sa'
6150052     //
6150053     sa.sin_family = AF_INET;
6150054     sa.sin_port = htons (atoi (port));
6150055     inet_pton (AF_INET, addr, &sa.sin_addr.s_addr);
6150056     //
6150057     //
6150058     // Open the socket.
6150059     //
6150060     sfdn = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
6150061     if (sfdn < 0)
6150062     {
6150063         perror (NULL);
6150064         return (5);
6150065     }
6150066     //
6150067     // Connect the 'sa' destination
6150068     //
6150069     status =
6150070     connect (sfdn, (struct sockaddr *) &sa, sizeof (sa));
6150071     if (status < 0)
6150072     {
6150073         perror (NULL);
6150074         close (sfdn);
6150075         return (7);
6150076     }
6150077     //
6150078     // Will read from the standard input and send to the
6150079     // other
6150080     // side.
6150081     //
6150082     while (1)
6150083     {
6150084         read_size = read (STDIN_FILENO, buffer, BUFSIZ);
6150085         if (read_size < 0)
6150086         {
6150087             perror (NULL);
6150088             close (sfdn);
6150089             return (8);
6150090         }
6150091         if (read_size == 0)
6150092         {
6150093             close (sfdn);
6150094             return (0);
6150095         }
6150096         //
6150097         sent_size =
6150098         send (sfdn, &buffer, (size_t) read_size, 0);
6150099         if (sent_size < 0)
6150100         {
6150101             perror (NULL);
6150102             close (sfdn);
6150103             return (9);
6150104         }
6150105         printf ("sent %i bytes\n", (int) sent_size);
6150106     }
6150107     //
6150108     // All done.
6150109     //
6150110     return (0);
6150111 }
6150112
6150113 -----
6150114 static void
6150115 usage (void)
6150116 {
6150117     fprintf (stderr, "Usage: tx_udp DEST_ADDR DEST_PORT\n");
6150118 }

```

### 96.1.51 applic/touch.c

« Si veda la sezione 86.26.

```

6160001 #include <fcntl.h>
6160002 #include <sys/stat.h>
6160003 #include <utime.h>
6160004 #include <stddef.h>

```

```

610005 #include <unistd.h>
610006 #include <errno.h>
610007 //-----
610008 static void usage (void);
610009 //-----
610010 int
610011 main (int argc, char *argv[], char *envp[])
610012 {
610013     int a;          // Argument index.
610014     int status;
610015     struct stat file_status;
610016     //
610017     // No options are known, but at least an argument
610018     // must be given.
610019     //
610020     if (argc < 2)
610021     {
610022         usage ();
610023         return (1);
610024     }
610025     //
610026     // Scan arguments.
610027     //
610028     for (a = 1; a < argc; a++)
610029     {
610030         //
610031         // Verify if the file exists, through the return
610032         // value of
610033         // 'stat()'. No other checks are made.
610034         //
610035         if (stat (argv[a], &file_status) == 0)
610036         {
610037             //
610038             // File exists: should be updated the times.
610039             //
610040             status = utime (argv[a], NULL);
610041             if (status != 0)
610042             {
610043                 perror (NULL);
610044                 return (2);
610045             }
610046         }
610047         else
610048         {
610049             //
610050             // File does not exist: should be created.
610051             //
610052             status =
610053                 open (argv[a],
610054                     O_WRONLY | O_CREAT | O_TRUNC, 0666);
610055             //
610056             if (status >= 0)
610057             {
610058                 //
610059                 // Here, the variable 'status' is the
610060                 // file
610061                 // descriptor to be closed.
610062                 //
610063                 status = close (status);
610064                 if (status != 0)
610065                 {
610066                     perror (NULL);
610067                     return (3);
610068                 }
610069             }
610070             else
610071             {
610072                 perror (NULL);
610073                 return (4);
610074             }
610075         }
610076     }
610077     return (0);
610078 }
610079 //-----
610080 static void
610081 usage (void)
610082 {
610083     fprintf (stderr, "Usage: touch FILE...\n");
610084 }
610085

```

## 96.1.52 applic/tty.c

Si veda la sezione 86.27.

```

617001 #include <fcntl.h>
617002 #include <sys/stat.h>
617003 #include <utime.h>
617004 #include <stddef.h>
617005 #include <unistd.h>
617006 #include <errno.h>
617007 #include <sys/os32.h>
617008 #include <sys/types.h>
617009 //-----
617010 static void usage (void);
617011 //-----
617012 int
617013 main (int argc, char *argv[], char *envp[])
617014 {
617015     int dev_minor;
617016     struct stat file_status;
617017     //
617018     // No options and no arguments.
617019     //
617020     if (argc > 1)
617021     {
617022         usage ();
617023         return (1);
617024     }
617025     //
617026     // Verify the standard input.
617027     //
617028     if (fstat (STDIN_FILENO, &file_status) == 0)
617029     {
617030         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
617031         {
617032             dev_minor = minor (file_status.st_rdev);
617033             //
617034             // If minor is equal to 0xFF, it is
617035             // '/dev/console'
617036             // that is not a controlling terminal, but
617037             // just
617038             // a reference for the current virtual
617039             // console.
617040             //
617041             if (dev_minor < 0xFF)
617042             {
617043                 printf ("/dev/console%i\n", dev_minor);
617044             }
617045         }
617046     }
617047     else
617048     {
617049         perror ("Cannot get standard input file status");
617050         return (2);
617051     }
617052     //
617053     return (0);
617054 }
617055 //-----
617056 static void
617057 usage (void)
617058 {
617059     fprintf (stderr, "Usage: tty\n");
617060 }
617061

```

## 96.1.53 applic/umount.c

Si veda la sezione 92.7.

```

618001 #include <unistd.h>
618002 #include <stdlib.h>
618003 #include <sys/stat.h>
618004 #include <sys/types.h>
618005 #include <fcntl.h>
618006 #include <errno.h>
618007 #include <signal.h>
618008 #include <stdio.h>
618009 #include <sys/wait.h>
618010 #include <stdio.h>
618011 #include <string.h>
618012 #include <limits.h>
618013 #include <sys/os32.h>
618014 //-----
618015 static void usage (void);
618016 //-----
618017 int
618018 main (int argc, char *argv[], char *envp[])

```

```

6180019 {
6180020     int status;
6180021     //
6180022     // One argument is mandatory.
6180023     //
6180024     if (argc != 2)
6180025     {
6180026         usage ();
6180027         return (1);
6180028     }
6180029     //
6180030     // System call.
6180031     //
6180032     status = umount (argv[1]);
6180033     if (status != 0)
6180034     {
6180035         perror (argv[1]);
6180036         return (2);
6180037     }
6180038     //
6180039     return (0);
6180040 }
6180041 //-----
6180042 static void
6180043 usage (void)
6180044 {
6180045     fprintf (stderr, "Usage: umount MOUNT_POINT\n");
6180046 }
6180047 }

```

### 96.1.54 applic/yes.c

« Si veda la sezione 86.28.

```

6190001 #include <stdio.h>
6190002 //-----
6190003 int
6190004 main (int argc, char *argv[], char *envp[])
6190005 {
6190006     int i;
6190007     //
6190008     if (argc > 1)
6190009     {
6190010         while (1)
6190011         {
6190012             printf ("%s", argv[1]);
6190013             for (i = 2; i < argc; i++)
6190014             {
6190015                 printf (" %s", argv[i]);
6190016             }
6190017             printf ("\n");
6190018         }
6190019     }
6190020     else
6190021     {
6190022         while (1)
6190023         {
6190024             printf ("y\n");
6190025         }
6190026     }
6190027     return (0);
6190028 }

```



## Indice analitico del volume

«

aaa 191 aaa.c 940 abort() 247 abort.c 846 abs() 247  
abs.c 846 accept() 203 accept.c 894 access() 248  
access.c 916 address resolution protocol 77 addr\_t 104 135  
allocated 191 allocated.c 940 applic.sep.ld 404  
arp 321 ARP 77 arp.c 941 arp.h 167 329 647  
arp\_clean() 168 arp\_clean.c 648 arp\_index() 168  
arp\_index.c 648 arp\_init() 168 arp\_init.c 649  
arp\_print.c 649 arp\_public.c 649 arp\_reference() 168  
arp\_reference.c 649 arp\_request() 168  
arp\_request.c 650 arp\_rx() 168 arp\_rx.c 651  
asctime() 252 asctime.c 908 assert() 248 assert.h 753  
ATA 35 ata.h 330 431 ata0 311 ata1 311 ata2 311  
ata3 311 ata4 311 ata5 311 ata6 311 ata7 311  
ata\_cmd\_identify\_device() 148  
ata\_cmd\_identify\_device.c 434  
ata\_cmd\_read\_sectors() 148  
ata\_cmd\_read\_sectors.c 434  
ata\_cmd\_write\_sectors() 148  
ata\_cmd\_write\_sectors.c 435 ata\_device() 148  
ata\_device.c 436 ata\_drq() 148 ata\_drq.c 437  
ata\_init() 147 ata\_init.c 438 ata\_lba28() 148  
ata\_lba28.c 441 ata\_public.c 441 ata\_rdy() 148  
ata\_rdy.c 442 ata\_read\_sector() 149 ata\_reset() 147  
ata\_reset.c 442 ata\_sector\_t 104 147 ata\_t 104 147  
ata\_valid() 147 ata\_valid.c 443  
ata\_write\_sector() 149 atexit() 249 atexit.c 847  
atoi() 250 atoi.c 847 atol() 250 atol.c 848 avvio 185  
basename() 250 basename.c 781 bbb.c 942 bind() 204  
bind.c 895 blk.h 138 332 416 blk\_ata() 332 blk\_ata.c 138 417  
blk\_cache\_check() 332 blk\_cache\_check.c 418 blk\_cache\_init() 333  
blk\_cache\_init.c 138 419 blk\_cache\_read() 333 blk\_cache\_read.c 138 419  
blk\_cache\_save() 333 blk\_cache\_save.c 138 419 blk\_cache\_t 104  
blk\_public.c 420 bochs 405 brk() 205 brk.c 916 build.h 628  
cat 192 cat.c 942 ccc.c 943 chdir() 206 chdir.c 917 chgrp 192  
chgrp.c 944 chiamata di sistema 124 chmod 192 chmod() 207  
chmod.c 901 945 chown 193 chown() 208 chown.c 917 946 clearerr() 251  
clearerr.c 797 CLI 12 cli() 113 cli.s 535 clock() 208  
clock.c 909 clock\_t.h 754 close() 209 close.c 917  
closedir() 252 closedir.c 765 conclusione 185 condotto 157  
connect() 209 connect.c 895 console 311 console0 311  
console1 311 console2 311 console3 311 cp 193 cp.c 947  
CPL 9 creat() 252 creat.c 774 crt0.mer.s 949 crt0.s 108 628  
crt0.sep.s 951 ctime() 252 ctype.h 754 date 193 date.c 953  
dev.h 137 334 420 dev\_ata() 338 dev\_ata.c 421 DEV\_CONSOLE 139  
DEV\_CONSOLEn 139 dev\_dm() 337 dev\_dm.c 137 422 DEV\_DMmn 139  
dev\_io() 137 337 dev\_io.c 137 423 dev\_kmem() 338 dev\_kmem.c 137 423  
DEV\_KMEM\_ARP 139 DEV\_KMEM\_FILE 139 DEV\_KMEM\_INODE 139  
DEV\_KMEM\_MMP 139 DEV\_KMEM\_NET 139 DEV\_KMEM\_PS 139  
DEV\_KMEM\_ROUTE 139 DEV\_KMEM\_SB 139 DEV\_MEM 139 dev\_mem() 339  
dev\_mem.c 426 DEV\_NULL 139 DEV\_PORT 139 DEV\_TTY 139  
dev\_tty() 339 dev\_tty.c 137 427 DEV\_ZERO 139 DIR.c 765  
directory\_t 104 dirent.h 764 dirname() 250 dirname.c 782  
div() 254 div.c 848 dm.h 340 429 dm\_init.c 430 dm\_public.c 431  
dm\_t 147 DPL 9 dup() 210 dup.c 918 dup2() 210 dup2.c 918  
eccezione 29 ed 194 ed.c 955 elf-to-os32 405 endgrent() 270  
endpwent() 275 environ 319 environ.c 918 environment.c 848  
errfn 255 errln 255 errno 255 errno.c 773 errno.h 768  
errset() 255 execl() 259 execl.c 918 execl() 259  
execlp.c 919 execlp() 259 execlp.c 919 execv() 259  
execv.c 920 execve() 211

execve.c 920 execvp() 259 execvp.c 921 exit() 249  
 exit.c 849 fchdir() 206 fchdir.c 922 fchmod() 207  
 fchmod.c 901 fchown() 208 fchown.c 922 fclose() 260  
 fclose.c 797 fcntl() 212 fcntl.c 775 fcntl.h 773  
 fdisk 406 fd\_dup() 163 343 fd\_dup.c 480  
 fd\_reference() 163 344 fd\_reference.c 481 fd\_t 104  
 feof() 261 feof.c 797 ferrord() 261 ferrord.c 797  
 fflush() 262 fflush.c 798 fgetc() 262 fgetc.c 798  
 fgetpos() 263 fgetpos.c 798 fgets() 263 fgets.c 798  
 FILE.c 796 fileno() 264 fileno.c 799  
 file\_image\_functions 407 file\_pipe\_make() 345  
 file\_pipe\_make.c 482 file\_reference() 159 345  
 file\_reference.c 482 file\_stdio\_dev\_make() 159  
 346 file\_stdio\_dev\_make.c 483 file\_t 104 159  
 fopen() 264 fopen.c 799 fork() 214 fork.c 922 format  
 410 fprintf() 286 fprintf.c 800 fputc() 266 fputc.c  
 800 fputs() 266 fputs.c 801 fread() 267 fread.c 801  
 free() 280 free.c 861 freopen() 264 freopen.c 801  
 fs.h 149 340 475 fsconf() 292 fsconf.c 802 fseek() 267  
 267 fseek.c 802 fseeko() 267 fseeko.c 802 fsetpos() 263  
 263 fsetpos.c 802 fstat() 236 fstat.c 902 fs\_init() 344  
 344 fs\_init.c 483 fs\_public.c 484 ftell() 268  
 ftell.c 803 ftello() 268 ftello.c 803 fwrite() 269  
 fwrite.c 803 GDT 14 gdt() 114 gdt.c 536 gdt\_load() 114  
 114 gdt\_load.s 536 gdt\_print() 114 gdt\_print.c 537  
 gdt\_public.c 537 gdt\_segment() 114 gdt\_segment.c 537  
 537 gdt\_t 104 getc() 262 getchar() 262 getchar.c 803  
 getcwd() 214 getcwd.c 923 getegid() 215 getegid.c  
 923 getenv() 269 getenv.c 850 geteuid() 216  
 geteuid.c 924 getgid() 215 getgid.c 924 getgrent() 270  
 270 getgrgid() 271 getgrnam() 271 getopt() 272  
 getopt.c 924 getpgrp() 216 getpgrp.c 927 getpid() 216  
 216 getpid.c 927 getppid() 216 getppid.c 928  
 getpwent() 275 getpwnam() 276 getpwuid() 276  
 gets() 263 gets.c 804 getty 321 getty.c 972 getuid() 216  
 216 getuid.c 928 global\_descriptor\_table 14 gmtime() 252  
 gmtime.c 909 grent.c 776 group 317 grp.h 776  
 header\_t 104 htonl() 251 htonl.c 761 htons() 251  
 htons.c 762 http 322 http.c 973 h\_addr\_t 104 168  
 ibm\_i386.h 112 368 530 ICMP 83 icmp.h 371 653 783  
 icmp\_rx() 172 icmp\_rx.c 653 icmp\_tx() 172  
 172 icmp\_tx.c 655 icmp\_tx\_echo() 172 icmp\_tx\_echo.c  
 656 icmp\_tx\_unreachable() 172  
 icmp\_tx\_unreachable.c 656 IDE 35 IDT 21 idt() 115  
 idt.c 538 idtr\_t 104 idt\_descriptor() 115  
 idt\_descriptor.c 539 idt\_irq\_remap() 115  
 115 idt\_irq\_remap.c 539 idt\_load() 115 idt\_load.s 540  
 idt\_print() 115 idt\_print.c 540 idt\_public.c 541  
 idt\_t 104 imaxabs() 247 imaxabs.c 781 imaxdiv() 254  
 imaxdiv.c 781 in.h 785 INB 11 inet.h 761 inet\_ntop() 277  
 277 inet\_ntop.c 762 inet\_pton() 278 inet\_pton.c  
 762 init 322 init.c 980 inittab 317 inode\_alloc() 154 346  
 154 346 inode\_alloc.c 484 inode\_check() 154 347  
 154 347 inode\_check.c 486 inode\_dir\_empty() 154 348  
 154 348 inode\_dir\_empty.c 487 inode\_file\_read() 154 349  
 154 349 inode\_file\_read.c 488 inode\_file\_write() 154 350  
 154 350 inode\_file\_write.c 490 inode\_free() 154 350  
 154 350 inode\_free.c 491 inode\_fzones\_read() 154 351  
 154 351 inode\_fzones\_read.c 492 inode\_fzones\_write() 154  
 351 351 inode\_fzones\_write.c 492 inode\_get() 154 352  
 154 352 inode\_get.c 493 inode\_pipe\_make() 154 353  
 154 353 inode\_pipe\_make.c 496 inode\_pipe\_read() 154 353  
 154 353 inode\_pipe\_read.c 497 inode\_pipe\_write() 154 354  
 154 354 inode\_pipe\_write.c 498 inode\_print() 154 354  
 154 354 inode\_print.c 499 inode\_put() 154 355 inode\_put.c  
 500 500 inode\_reference() 154 355 inode\_reference.c  
 501 501 inode\_save() 154 356 inode\_save.c

503 inode\_stdio\_dev\_make() 154 357  
 inode\_stdio\_dev\_make.c 503 inode\_t 104 152  
 inode\_truncate() 154 357 inode\_truncate.c 504  
 inode\_zone() 154 358 inode\_zone.c 506  
 input\_line() 278 input\_line.c 885 interfaccia di rete 56  
 interrupt\_descriptor\_table 21 interruzione 24 inttypes.h 777  
 in\_16() 112 in\_8() 112 ip.h 168 371 657 786 ipconfig  
 323 ipconfig() 217 ipconfig.c 887 983 IPv4 76 79  
 ip\_checksum() 168 ip\_checksum.c 658 ip\_header() 168  
 168 ip\_header.c 659 ip\_mask() 168 ip\_mask.c 659  
 ip\_public.c 660 ip\_reference() 168  
 ip\_reference.c 660 ip\_rx() 168 ip\_rx.c 660  
 ip\_table[] 169 ip\_tx() 168 ip\_tx.c 663 IRET 24  
 irq\_off() 113 irq\_off.c 541 irq\_on() 113 irq\_on.c  
 541 isatty() 279 isatty.c 928 isr.s 117 542  
 isr\_exception\_name() 116 isr\_exception\_name.c  
 552 isr\_exception\_unrecoverable() 116  
 116 isr\_exception\_unrecoverable.c 552  
 isr\_irq\_clear() 116 isr\_irq\_clear.c 553  
 isr\_irq\_clear\_pic1() 116 isr\_irq\_clear\_pic1.c  
 553 isr\_irq\_clear\_pic2() 116  
 isr\_irq\_clear\_pic2.c 553 isr\_n() 116 issue 317 I&D  
 100 kbd.h 372 443 kbd\_isr() 144 kbd\_isr.c 443  
 kbd\_load() 144 kbd\_load.c 445 kbd\_public.c 447  
 kbd\_t 104 kernel.ld 108 410 kill 195 kill() 217  
 kill.c 794 984 kmain.c 629 kmem\_arp 312 kmem\_file  
 312 kmem\_inode 312 kmem\_mmp 312 kmem\_net 313  
 kmem\_ps 313 kmem\_route 313 kmem\_sb 314 k\_exit.s 554  
 k\_gets.c 554 k\_perror.c 555 k\_printf.c 555  
 k\_sleep.c 555 k\_stime.c 556 k\_usleep.c 556  
 k\_vprintf.c 557 k\_vsprintf.c 557 labs() 247 labs.c  
 851 ldiv() 254 ldiv.c 851 LGDT 20 libgen.h 781  
 lib\_k.h 372 554 lib\_s.h 372 558 LIDT 23 limits.h 754  
 link() 218 link.c 929 listen() 219 listen.c 896  
 llabs() 247 llabs.c 851 lldiv() 254 lldiv.c 851 ln  
 195 ln.c 987 localtime() 252 login 196 login.c 989  
 longjmp() 129 231 longjmp.c 791 ls 196 ls.c 991  
 lseek() 220 lseek.c 929 main() 109 main.h 373 627  
 major() 280 major.c 904 MAKEDEV 323 makedev() 280  
 makedev.c 904 MAKEDEV.c 938 makeit.sep 411  
 malloc() 280 malloc.c 862 man 197 man.c 995  
 mboot\_cmdline\_opt() 107 mboot\_cmdline\_opt.c 642  
 mboot\_public.c 643 mboot\_save() 107 mboot\_save.c  
 643 mb\_alloc() 135 mb\_alloc.c 636 mb\_alloc\_size() 135  
 135 mb\_alloc\_size.c 637 mb\_clean() 135 mb\_clean.c  
 638 mb\_free() 135 mb\_free.c 638 mb\_print() 135  
 135 mb\_print.c 639 mb\_public.c 640 mb\_reduce() 135  
 135 mb\_reduce.c 640 mb\_reference() 135  
 135 mb\_reference.c 641 mb\_size() 135 mb\_size.c 641 mem  
 314 memccpy() 281 memccpy.c 867 memchr() 281  
 memchr.c 868 memcmp() 282 memcmp.c 868 memcpy() 282  
 memcpy.c 868 memmove() 283 memmove.c 868 memory.c  
 135 memory.h 135 373 635 memset() 283 memset.c 869  
 MEM\_BLOCK\_SIZE 135 MEM\_MAX\_BLOCKS 135 menu.c 634  
 minor() 280 minor.c 904 mkdir 197 mkdir() 220  
 mkdir.c 902 999 mknod() 221 mknod.c 903 mktime() 252  
 mktime.c 911 mmcheck 198 mmcheck.c 1002 more 198  
 more.c 1004 mount 324 mount() 222 mount.c 888 1007  
 multiboot.h 374 641 multiboot\_t 104 multiboot  
 specification 105 namep() 283 namep.c 888 nc 199 nc.c  
 1008 NE2000 56 ne2k.h 374 447 ne2k\_check() 164  
 ne2k\_check.c 449 ne2k\_isr() 164 ne2k\_isr.c 450  
 ne2k\_isr\_expect() 164 ne2k\_isr\_expect.c 451  
 ne2k\_reset() 164 ne2k\_reset.c 452 ne2k\_rx() 164  
 ne2k\_rx.c 457 ne2k\_rx\_reset() 164  
 ne2k\_rx\_reset.c 461 ne2k\_tx() 164 ne2k\_tx.c 462  
 net.h 164 375 644 net\_buffer\_eth() 166

net\_buffer\_eth.c 665 net\_buffer\_lo() 166  
net\_buffer\_lo.c 665 net\_eth\_ip\_tx() 166  
net\_eth\_ip\_tx.c 666 net\_eth\_tx() 166  
net\_eth\_tx.c 668 net\_index() 166 net\_index.c 668  
net\_index\_eth() 166 net\_index\_eth.c 668  
net\_init() 166 net\_init.c 669 net\_print.c 672  
net\_public.c 672 net\_rx() 166 net\_rx.c 673 NIC 56  
ntohl() 251 ntohl.c 764 ntohs() 251 ntohs.c 764 null  
315 NULL.h 753 offsetof() 284 open() 223 open.c 775  
opendir() 285 opendir.c 766 os32 97 os32.h 137 877  
OUTB 11 out\_16() 112 out\_8() 112 part.h 376 703  
passwd 318 PATA 35 path\_device() 161 365  
path\_device.c 512 path\_fix() 161 365 path\_fix.c  
513 path\_full() 161 366 path\_full.c 514  
path\_inode() 161 366 path\_inode.c 514  
path\_inode\_link() 161 367 path\_inode\_link.c 517  
PCI 48 pci.h 376 463 pci\_init.c 465 pci\_public.c 466  
*peripheral component interconnect* 48 perror() 285 perror.c  
804 PIC 30 ping 324 ping.c 1012 pipe 157 pipe() 225  
pipe.c 929 PIT 33 port 315 printf() 286 printf.c 805  
proc.h 117 377 704 proc\_available() 377  
proc\_available.c 706 proc\_dump\_memory() 377  
proc\_dump\_memory.c 707 proc\_init() 125 378  
proc\_init.c 708 proc\_print() 379 proc\_print.c 710  
proc\_public.c 712 proc\_reference() 379  
proc\_reference.c 712 proc\_scheduler() 126 381  
proc\_scheduler.c 719 proc\_sch\_net() 379  
proc\_sch\_net.c 712 proc\_sch\_signals() 380  
proc\_sch\_signals.c 713 proc\_sch\_terminals() 380  
proc\_sch\_terminals.c 714 proc\_sch\_timers() 381  
proc\_sch\_timers.c 719 proc\_sig\_chld() 383  
proc\_sig\_chld.c 722 proc\_sig\_cont() 383  
proc\_sig\_cont.c 722 proc\_sig\_core() 384  
proc\_sig\_core.c 723 proc\_sig\_handler() 127 384  
proc\_sig\_handler.c 724 proc\_sig\_ignore() 385  
proc\_sig\_ignore.c 727 proc\_sig\_off() 386  
proc\_sig\_off.c 727 proc\_sig\_on() 386  
proc\_sig\_on.c 727 proc\_sig\_status() 386  
proc\_sig\_status.c 727 proc\_sig\_stop() 387  
proc\_sig\_stop.c 727 proc\_sig\_term() 387  
proc\_sig\_term.c 728 proc\_sys\_exec() 388  
proc\_sys\_exec.c 728 proc\_t 104 121  
proc\_timer\_init() 389 proc\_timer\_init.c  
738 proc\_wakeup\_pipe\_read()  
390 proc\_wakeup\_pipe\_read.c  
739 proc\_wakeup\_pipe\_write()  
390 proc\_wakeup\_pipe\_write.c  
739 proc\_wakeup\_terminal() 390  
proc\_wakeup\_terminal.c 739 *programmable interrupt  
controller* 30 *programmable interval timer* 33 ps 199 ps.c 1015  
PS/2 34 ptr() 391 ptr.c 740 ptrdiff\_t.h 755 putc() 266  
putc() 266 putchar.c 805 putenv() 289  
putenv.c 851 puts() 266 puts.c 805 pwd.h 788 pwent.c  
789 qemu 415 qsort() 289 qsort.c 853 rand() 290  
rand.c 854 read() 226 read.c 930 readdir() 291  
readdir.c 767 realloc() 280 realloc.c 865  
recvfrom() 226 recvfrom.c 896 restrict.h 755  
rewind() 291 rewind.c 806 rewinddir() 292  
rewinddir.c 768 rm 200 rm.c 1018 rmdir 201 rmdir()  
227 rmdir.c 931 1019 route 325 route.c 1020 route.h  
392 674 routeadd() 228 routeadd.c 890 routedel()  
229 routedel.c 891 route\_init() 171 route\_init.c  
674 route\_print.c 675 route\_public.c  
675 route\_remote\_to\_local()  
171 route\_remote\_to\_local.c  
676 route\_remote\_to\_router() 171  
route\_remote\_to\_router.c 676 route\_sort() 171

route\_sort.c 677 RPL 9 run.c 634 sa\_family\_t.h 893  
sbrk() 205 sbrk.c 931 sb\_inode\_status() 150 359  
sb\_inode\_status.c 521 sb\_mount() 150 359  
sb\_mount.c 521 sb\_print() 150 360 sb\_print.c 524  
sb\_reference() 150 360 sb\_reference.c 524  
sb\_save() 150 361 sb\_save.c 525 sb\_t 104 149  
sb\_zone\_status() 150 359 sb\_zone\_status.c 526  
scanf() 292 scanf.c 806 screen.h 393 466  
screen\_cell() 145 screen\_clear() 145  
screen\_clear.c 467 screen\_current() 145  
screen\_current.c 467 screen\_init() 145  
screen\_init.c 467 screen\_newline() 145  
screen\_new\_line.c 468 screen\_number() 145  
screen\_number.c 468 screen\_pointer() 145  
screen\_pointer.c 469 screen\_public.c 469  
screen\_putc() 145 screen\_putc.c 469  
screen\_scroll() 145 screen\_scroll.c 470  
screen\_select() 145 screen\_select.c 470 screen\_t  
104 screen\_update() 145 screen\_update.c 471  
SEEK.h 753 *selector* 10 *selettore* 10 send() 229 send.c 898  
setbuf() 296 setbuf.c 806 setegid() 230 setegid.c  
932 setenv() 296 setenv.c 855 seteuid() 233  
seteuid.c 932 setgid() 230 setgid.c 932 setgrent()  
270 setjmp() 129 231 setjmp.h 790 setjmp.s 791  
setpgrp() 233 setpgrp.c 932 setpwnent() 275  
setuid() 233 setuid.c 933 setvbuf() 296 setvbuf.c  
806 shell 201 shell.c 1021 signal() 234 signal.c 794  
signal.h 792 size\_t.h 756 sleep() 235 sleep.c 933  
snprintf() 286 snprintf.c 806 socket 319 socket()  
236 socket.c 899 socket.h 893 socklen\_t.h 899  
sock\_free\_port() 362 sock\_free\_port.c 526  
sock\_reference() 362 sock\_reference.c 526 *specifiche  
multiboot* 105 sprintf() 286 sprintf.c 806 srand() 290  
sscanf() 292 sscanf.c 807 stack.s 108 635 stat() 236  
stat.c 903 stat.h 899 stdarg.h 756 stdbool.h 756  
stddef.h 756 stdint.h 756 stdio.h 297 794 stdlib.h  
844 STI 12 sti() 113 sti.s 553 stime() 241 stime.c 913  
strcat() 298 strcat.c 869 strchr() 299 strchr.c 869  
strcmp() 299 strcmp.c 870 strcoll() 299 strcoll.c  
870 strcpy() 300 strcpy.c 870 strcspn() 302  
strcspn.c 870 strdup() 300 strdup.c 871 strerror()  
301 strerror.c 871 string.h 866 strlen() 301  
strlen.c 873 strncat() 298 strncat.c 873 strncmp()  
299 strncmp.c 873 strncpy() 300 strncpy.c 873  
strpbrk() 302 strpbrk.c 874 strrchr() 299  
strrchr.c 874 strspn() 302 strspn.c 874 strstr()  
303 strstr.c 875 strtok() 303 strtok.c 875 strtol()  
304 strtol.c 856 strtoul() 304 strtoul.c 859  
strxfrm() 305 strxfrm.c 877 sys() 239 sys.s 891  
syslinux 416 sysroutine() 124 125 391 sysroutine.c  
740 s\_accept.c 562 s\_bind.c 564 s\_brk() 372 s\_brk.c  
565 s\_chdir() 162 372 s\_chdir.c 569 s\_chmod() 162 372  
s\_chmod.c 569 s\_chown() 162 163 372 s\_chown.c 570  
s\_clock() 372 s\_clock.c 571 s\_close() 372  
s\_close.c 571 s\_connect.c 572 s\_dup() 163 372  
s\_dup.c 575 s\_dup2() 163 372 s\_dup2.c 575  
s\_fchmod() 163 372 s\_fchmod.c 575 s\_fchown() 372  
s\_fchown.c 576 s\_fcntl() 163 372 s\_fcntl.c 577  
s\_fork() 372 s\_fork.c 578 s\_fstat() 163 372  
s\_fstat.c 582 s\_ipconfig.c 583 s\_kill() 372  
s\_kill.c 584 s\_link() 162 372 s\_link.c 585  
s\_listen.c 586 s\_longjmp() 129 372 s\_longjmp.c 587  
s\_lseek() 163 372 s\_lseek.c 588 s\_mkdir() 162 372  
s\_mkdir.c 589 s\_mknod() 162 372 s\_mknod.c 591  
s\_mount() 162 372 s\_mount.c 592 s\_open() 162 372  
s\_open.c 592 s\_pipe() 163 372 s\_pipe.c 596 s\_read()  
163 372 s\_read.c 598 s\_recvfrom.c 600 s\_routeadd.c

606 s\_routedel.c 607 s\_sbrk() 372 s\_sbrk.c 608  
s\_send.c 609 s\_setegid() 372 s\_setegid.c 612  
s\_seteuid() 372 s\_seteuid.c 612 s\_setgid() 372  
s\_setgid.c 613 s\_setjmp() 129 372 s\_setjmp.c 613  
s\_setuid() 372 s\_setuid.c 614 s\_signal() 372  
s\_signal.c 614 s\_socket.c 615 s\_stat() 162 372  
s\_stat.c 617 s\_stime() 372 s\_stime.c 618  
s\_tcgetattr() 372 s\_tcgetattr.c 618  
s\_tcsetattr() 372 s\_tcsetattr.c 619 s\_time() 372  
s\_time.c 620 s\_umount() 162 372 s\_umount.c 620  
s\_unlink() 162 372 s\_unlink.c 622 s\_wait() 372  
s\_wait.c 624 s\_write() 163 372 s\_write.c 625  
s\_\_exit() 372 s\_\_exit.c 560 tap0 416 tastiera 34  
tcgetattr() 239 tcgetattr.c 906 TCP 87 tcp() 174  
tcp.c 679 tcp.h 394 679 787 tcp\_close() 174  
tcp\_close.c 690 tcp\_connect() 174 tcp\_connect.c  
691 tcp\_rx\_ack() 174 tcp\_rx\_ack.c 692  
tcp\_rx\_data() 174 tcp\_rx\_data.c 693 tcp\_show() 174  
tcp\_show.c 695 tcp\_status.c 695 tcp\_test.c 696  
tcp\_tx\_ack() 174 tcp\_tx\_ack.c 697 tcp\_tx\_raw() 174  
tcp\_tx\_raw.c 698 tcp\_tx\_rst() 174  
tcp\_tx\_rst.c 699 tcp\_tx\_sock() 174 tcp\_tx\_sock.c  
700 tcsetattr() 239 tcsetattr.c 907 termios.h 905  
time() 241 time.c 913 time.h 907 time\_t.h 758 touch  
202 touch.c 1039 tty 202 315 tty.c 1041 tty.h 396 472  
ttyname() 306 ttyname.c 933 tty\_console() 142  
tty\_console.c 472 tty\_init() 142 tty\_init.c 473  
tty\_public.c 474 tty\_read() 142 tty\_read.c 474  
tty\_reference() 142 tty\_reference.c 475 tty\_t 104  
tty\_write() 142 tty\_write.c 475 types.h 904  
t\_fcntl.c 1024 t\_fifo.c 1024 t\_grp.c 1026 t\_nc.c  
1026 t\_ping2.c 1030 t\_pipe.c 1031 t\_read.c 1032  
t\_ret.c 1033 t\_rx\_udp.c 1033 t\_scr.c 1034  
t\_setjmp.c 1035 t\_sig.c 1035 t\_sig2.c 1036  
t\_tx\_tcp.c 1037 t\_tx\_udp.c 1038 UDP 85 udp.h 702 788  
udp\_tx() 174 udp\_tx.c 702 umask() 242 umask.c 903  
umount 324 umount() 222 umount.c 891 1041 unistd.h  
913 unlink() 243 unlink.c 934 unsetenv() 296  
unsetenv.c 859 utime.c 935 utime.h 935 u-area 121  
vfprintf() 307 vfprintf.c 807 vfscanf() 308  
vfscanf.c 807 vfscanf.c 807 VGA 13 vprintf() 307  
vprintf.c 827 vscanf() 308 vscanf.c 827  
vsnprintf() 307 vsnprintf.c 828 vsprintf() 307  
vsprintf.c 843 vsscanf() 308 vsscanf.c 844 wait() 243  
wait.c 905 wait.h 905 wchar\_t.h 758 write() 244  
write.c 934 yes 202 yes.c 1042 zero 315 zno\_t 104  
zone\_alloc() 152 363 zone\_alloc.c 527 zone\_free() 152 363  
zone\_free.c 528 zone\_print() 152 364  
zone\_print.c 529 zone\_read() 152 364 zone\_read.c  
529 zone\_write() 152 364 zone\_write.c 530  
z\_perror() 244 z\_perror.c 892 z\_printf() 244  
z\_printf.c 892 z\_vprintf() 244 z\_vprintf.c 892  
\_\_alloc\_list.c 860 \_Exit() 203 \_exit() 203 \_exit.c  
915 \_Exit.c 846 \_gcc.h 758 \_in\_16() 112 \_in\_16.s 533  
\_in\_32.s 534 \_in\_8() 112 \_in\_8.s 534 \_lldiv.c 759  
\_out\_16() 112 \_out\_16.s 534 \_out\_32.s 535 \_out\_8() 112  
\_out\_8.s 535 \_sighandler\_wrapper.s 793  
\_\_lldiv.c 760 \_\_divdi3.c 759 \_\_moddi3.c 759  
\_\_udivdi3.c 759 \_\_umoddi3.c 759